UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A217 114

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): NMSU - ECE - 89 - 005	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: New Mexico State University; 6b. OFFICE SYMBOL (If applicable): PARL	7a. NAME OF MONITORING ORGANIZATION: U. S. Army Research Office
6c. ADDRESS (City, State, and ZIP Code): Las Cruces, NM 88003	7b. ADDRESS (City, State, and ZIP Code): P. O. Box 12211 Research Triangle Park, NC 27709-2211
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: U. S. Army Research Office; 8b. OFFICE SYMBOL (If applicable):	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): P. O. Box 12211 Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
The Contour Model Architecture and Assembly Language

12. PERSONAL AUTHOR(S) John B. Johnston

13a. TYPE OF REPORT	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT

16. SUPPLEMENTARY NOTATION
The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this report is limited to presenting the tagged record structure of the Contour Model Architecture CMA, the static structure of individual snapshots of a nested module computation as realized in CMA, and the Contour Model Assembly Language CMAL. Pedagogic illustrations of the CMAL-level evolution of a nested module computation are given in a separate report.

The Contour Model CM as it currently exists can account for most of the semantic features of a broad spectrum of nested module computations; CM does not yet contain specific features for input/output, interrupts, or selectively restricted memory access. The architecture CMA and the assembly language CMAL together constitute a detailed operational mechanism which realized CM as it currently exists. CMA is a relatively conventional, fully tagged, stack-oriented architecture whose tagged record structure and assembly language are intended to be implemented in micocode.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

THE CONTOUR MODEL ARCHITECTURE AND ASSEMBLY LANGUAGE

JOHN B. JOHNSTON

NMSU-ECE-89-005
June 1989

Table of Contents

ABBREVIATIONS:

```
stk      = stack
lab      = labl (i-lab)
reg      = registers
val      = value
ep       = environment pointer
c-rvrt   = conditional duplication revert
```

00	rs	revert stk	038
01	cs	c-rvrt stk	038
02	ee	extract ep	038
03	rl	restore lab	038
04	dl	discard lab	038
05	rsee	revert stk, extract ep	038
06	csee	c-rvrt stk, extract ep	038
07	rsrl	revert stk, restore lab	038
08	csrl	c-rvrt stk, restore lab	038
09	rlrr	restore lab, restore reg	038
10	dlrr	discard lab, restore reg	038
11	rsrlrr	revert stk, restore lab, restore reg	039
12	csrlrr	c-rvrt stk, restore lab, restore reg	039
13	rvrs	retain val, revert stk	039
14	rvcs	retain val, c-rvrt stk	039
15	rvee	retain val, extract ep	039
16	rvrl	retain val, restore lab	039
17	rvdl	retain val, discard lab	039
18	rvrsee	retain val, revert stk, extract ep	039
19	rvcsee	retain val, c-rvrt stk, extract ep	039
20	rvrsrl	retain val, revert stk, restore lab	039
21	rvcsrl	retain val, c-rvrt stk, restore lab	040
22	rvrlrr	retain val, restore lab, restore reg	040
23	rvdlrr	retain val, discard lab, restore reg	040
24	rvrsrlrr	retain val, revert stk, restore lab, restore reg	040
25	rvcsrlrr	retain val, c-rvrt stk, restore lsb, restore reg	040

0. INTRODUCTION

The purpose of this report is limited to presenting the tagged record structure of the Contour Model Architecture CMA, the static structure of individual snapshots of a nested module computation as realized in CMA, and the Contour Model Assembly Language CMAL. Pedagogic illustrations of the CMAL-level evolution of a nested module computation are given in a separate report.

The Contour Model CM as it currently exists can account for most of the semantic features of a broad spectrum of nested module computations; CM does not yet contain specific features for input/output, interrupts, or selectively restricted memory access. The architecture CMA and the assembly language CMAL together constitute a detailed operational mechanism which realizes CM as it currently exists. CMA is a relatively conventional, fully tagged, stack-oriented architecture whose tagged record structure and assembly language are intended to be implemented in microcode. The architectural potency required of CMA to realize CM is contained in: the tagged record structure (sec. 1); the computation snapshot structure, including the record retention feature (sec. 2); and a few special instructions of the assembly language CMAL (sec. 3). CMA incorporates a segmented virtual memory system with provisions for MULTICS-like dynamic linking, but details of this memory system are treated in a separate report; for purposes of this report it suffices to envision memory as simply a large sequence of byte-size memory units, each individually addressable.

A nested module computation is a time-sequence of **snapshots**. Each snapshot is a data structure having both a **program component** and an **execution component**. In this report, the program component is treated as being time invariant throughout a computation; the course of execution of the program is recorded in the execution component. A data structure is composed of poly records and their mono record subrecords.

1. The Contour Model Architecture Tagged Record Stucture

There are two classes of records in CMA: **mono records**, which are elementary data items, and **poly records**, which are aggregate data items. Only poly records can be allocated; mono records can exist only as subrecords of poly records. Each record is a byte sequence which is the concatenation of two bit sequences: a **tag** and a **value**. The syntactic structure and byte length of a record are determined by the tag of the record and remain invariant during the lifetime of the record.

1.0. Mono Record Structure

A mono record is the concatenation of a tag and a value. The tag of a mono record consists simply of a fixed but unspecified bit pattern called the type field of the tag. The value of a mono record is a bit pattern which has a meaning only when interpreted relative to the tag of the mono record. Each mono record has associated with it a finite set of bit patterns which can serve as values of mono records having that type. The types, values, and syntactic structures of mono records are specified by the syntax for mono records displayed in **Table 1**.

The **null** mono record has a vacuous - that is, zero bit length - value, and a tag type field which is a byte consisting of eight 0 bits; null is the only mono record whose tag bit pattern is specified. There are only two distinct **logical** values: f (false) and t (true). The value of a **tag** mono record may be the tag of any mono record or the tag of any poly record; since tags of mono and poly records may be of various different lengths, the value of a tag record determines it's own length and hence that of the tag record. A **mask** mono record serves as a register selector for saving and restoring selected virtual processor registers to and from the virtual processor stack; the 16 mask value bits <msk.0>, ... , <msk.F> are in one-to-one correspondence with the 16 virtual processor registers R0, ... , RF, and serve to identify the registers whose contents are to be saved or restored. The **integer** values are bit patterns of some fixed but unspecified bit length which represent integers according to the 2's complement interpretation. The value of a **pointer** mono record is interpreted in such a manner that the pointer mono record in effect points to the tag of some poly record; the syntactic structure and interpretation of a pointer value are given in a separate report. The value of a **subpointer** mono record is the concatenation of a pointer value and an integer value which serves as a subrecord index; hardware use of such a subpointer value to access an indexed subrecord requires microcode interpretation of the tag of the poly record pointed to by the pointer value portion of the subpointer value. In this report we shall indicate pointer values in the symbolic form "@ P", where "P" is an identifier which designates the intended target poly record. A <cp> is a **contour-pointer**, that is, a pointer whose target poly record is a program contour. An <ip> is an **instruction-pointer**, that is, either a pointer which points to an instruction record or else a subpointer which points to a code byte subrecord of an instruction record. An <ep> is an **environment-pointer**, which either is null or else is a pointer whose target poly record is an execution contour; as discussed in sec. 2, an environment-pointer identifies what is known as an execution environment. A **contour-label** mono record thus designates both a program contour and an execution environment, while an **instruction-label** mono record designates both a code point within an instruction record and an execution environment. An <iden>, or identifier, is a non-empty sequence of letters and digits, the first of which is a capital letter.

```
00 <rec mono>              ::= <rec mmm>                                          mmm ε <type mono>
01 <rec mmm>               ::= <tag mmm> <val mmm>                                mmm ε <type mono>
02 <type mono>             ::= <type scaler> | <type struct>
03 <type scaler>           ::= null | log | tag | msk | int
04 <type struct>           ::= ptr | subptr | i-lab | ip-lab | cp-lab
05 <tag mono>              ::= <tag mmm>                                          mmm ε <type mono>
06 <tag mmm>               ::= mmm                                                mmm ε <type mono>
07 <val null>              ::= ε                                                  (ε means vacuous)
08 <val log>               ::= f | t                                                (false or true)
09 <val tag>               ::= <tag mono> | <tag poly>
10 <val msk>               ::= <msk.F><msk.E>...<msk.0>
11 <msk.i>                 ::= 0 | 1                                          0 ≤ i < vp.len = 16
12 <val int>               ::= <sign><magnitude>
13 <sign>                  ::= ε | + | -
14 <magnitude>             ::= <nv seq digit>
15 <nv seq digit>          ::= <digit> | <nv seq digit><digit>
16 <digit>                 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
17 <val ptr>               ::= @ <iden>
18 <val subptr>            ::= @ <iden>.<subrec desig>
19 <subrec desig>          ::= <val int> | <spec subrec desig> | <iden>
20 <spec subrec desig>     ::= len | ref | pid | lab | ip | ep | sp | dp | ap |
                               con | dsp | tos | xref | xlab
21 <iden>                  ::= <cap let><seq iden char>
22 <seq iden char>         ::= ε | <seq iden char><iden char>
23 <iden char>             ::= <cap let> | <small let> | <digit>
24 <cap let>               ::= A | ... | Z
25 <small let>             ::= a | ... | z
26 <val i-lab>             ::= <ip> <ep>                                  instruction label
27 <val ip-lab>            ::= <pip> <ep>                       instruction procedure label
28 <val cp-lab>            ::= <cp> <ep>                           contour procedure label
29 <ip>                    ::= <rec subptr>                           instruction pointer
30 <pip>                   ::= <rec ptr>                      instruction procedure pointer
31 <cp>                    ::= <rec ptr>                           contour procedure pointer
32 <ep>                    ::= <rec ptr> | null                          environment pointer
```

Table 1: A Syntax for CMA Mono Records

1.1. Poly Record Structure

A poly record is the concatenation of a tag and a value. The tag of a poly record consists of a fixed but unspecified bit pattern called the **type** field of the tag, zero or more special bits whose use depends on the type of poly record, and an embedded integer mono record called the **length** field which is program accessible as a read-only special subrecord. The value of a poly record consists of: an integer mono record called the **reference count** field which is program accessible as a read-only special subrecord of the poly record; zero or more additional special subrecords depending on the type of poly record; and zero or more non-special subrecords. The length special subrecord of a poly record equals the number of non-special subrecords of the poly record; this length is set by microcode during allocation of the poly record and remains fixed during the lifetime of the poly record. The reference count special subrecord of a poly record is maintained by microcode to reflect the number of (sub)pointers (in)to the poly record. Each special or non-special subrecord of a poly record begins at a byte position within the poly record which depends solely on the index of the subrecord and the tag of the poly record, and not on the type of the subrecord. Special subrecords of poly records have negative indexes, and are always mono records. Non-special subrecords of a poly record have ordinal (non-negative) indexes; depending on the type of the poly record, either all its non-special subrecords are mono records or else all its non-special subrecords are single bytes. The types and syntactic structures of poly records are specified by the syntax for poly records displayed in **Table 2**.

A **text** poly record has two special subrecords: the length field (index -2), and the reference count field (index -1). Each non-special subrecord of a text poly record is an 8-bit byte representing one of the 256 characters.

An **instruction** poly record has three special subrecords: the length field (index -3), and the reference count field (index -2), and the environment pointer subrecord (index -1). The non-special subrecords of an instruction poly record comprise a byte sequence which constitutes the coded form of a sequence of CMAL instructions.

```
00 <rec poly>          ::= <rec ppp>                                                    ppp ε <type poly>
01 <rec ppp>           ::= <tag ppp> <refval ppp>                                       ppp ε <type poly>
02 <type poly>         ::= <type poly byte> | <type poly mono>
03 <type poly byte>    ::= txt | ins
04 <type poly mono>    ::= stk | vct | p-con | e-con | vp
05 <tag poly>          ::= <tag ppp>                                                    ppp ε <type poly>
06 <tag ppp>           ::= ppp <ppp.len>                          p-con,e-con,vp, ≠ ppp ε <type poly>
07 <tag ppp>           ::= con <ppp.len>                                                ppp = p-con,e-con
08 <ppp.len>           ::= <rec int>                                        vp ≠ ppp ε <type poly>
09 <tag vp>            ::= vp <vp state>                                  (implicitly, vp.len = 16)
10 <vp state>          ::= new | awake | asleep | terminated
11 <refval ppp>        ::= <ppp.ref> <val ppp>                                          ppp ε <type poly>
12 <ppp.ref>           ::= <rec int>                                                    ppp ε <type poly>
13 <val txt>           ::= <txt.0><txt.1>...<txt.n>                                        n = txt.len-1
14 <txt.i>             ::= <char> (256 character set)                                   0 ≤ i < txt.len
15 <val ins>           ::= <ins.ep> <ins.0><ins.1>...<ins.n>                               n = ins.len-1
16 <ins.i>             ::= <byte>                                                       0 ≤ i < ins.len
17 <byte>              ::= <byte.0><byte.1>...<byte.7>
18 <byte.i>            ::= <bit>
19 <bit>               ::= 0 | 1
20 <val stk>           ::= <stk.tos> <stk.sp> <stk.0><stk.1>...<stk.n>                     n = stk.len-1
21 <stk.tos>           ::= <rec int>
22 <stk.sp>            ::= <sp>
23 <sp>                ::= null | <rec ptr>                                                stack ptr
24 <stk.i>             ::= <byte>                                                       0 ≤ i < stk.len
25 <val vct>           ::= <vct.0><vct.1>...<vct.n>                                        n = vct.len-1
26 <vct.i>             ::= <rec mono>                                                   0 ≤ i < vct.len
27 <val p-con>         ::= <p-con.dsp> <p-con.con> <p-con.ip> <p-con.ep>
                           <p-con.0> <p-con.1> ... <p-con.n>                            n = p-con.len-1
28 <p-con.ord>         ::= <rec int> | null
29 <p-con.tag>         ::= <rec tag>
30 <p-con.ip>          ::= <ip>                                                            see 1.0.27
31 <p-con.ep>          ::= <ep>                                                            see 1.0.28
32 <p-con.i>           ::= <rec mono>                                                   0 ≤ i < p-con.len
33 <val e-con>         ::= <e-con.dsp> <e-con.sp> <e-con.ap> <e-con.ep>
                           <e-con.0> <e-con.1> ... <e-con.n>                            n = e-con.len-1
34 <e-con.dsp>         ::= <rec mono>
35 <e-con.sp>          ::= <sp>                                                            see 1.1.23
36 <e-con.ap>          ::= <ap>
37 <ap>                ::= <rec ptr>                                                    antecedent ptr
38 <e-con.ep>          ::= <ep>                                                            see 1.0.28
39 <e-con.i>           ::= <rec mono>                                                   0 ≤ i < e-con.len
40 <val vp>            ::= <vp.pid> <vp.dp> <vp.sp> <vp.lab>
                           <vp.0> <vp.1> ... <vp.F>
41 <vp.pid>            ::= <rec ptr>                                        processor identity register
42 <vp.dp>             ::= null | <rec ptr>                                                display ptr
43 <vp.sp>             ::= <sp>                                                            see 1.1.23
44 <vp.lab>            ::= <rec i-lab>                                                  label register
45 <vp.i>              ::= <vp reg Ri>                                                working register Ri
46 <vp reg Ri>         ::= <rec mono>                                                   0 ≤ i < vp.len = 16
```

Table 2: A Syntax for CMA Poly Records

A stack poly record has four special subrecords: the length field (index -4), the reference count field (index -3), the top-of-stack subrecord (index -2), and the stack pointer subrecord (index -1). The non-special subrecords of a stack poly record comprise a byte sequence which constitutes the place of residence for the mono records which are pushed into and popped from the stack record. At all times, the top-of-stack integer has a non-negative value which does not exceed the value of the length field. The stack record is empty if and only if the top-of-stack integer equals the length field. When the stack record is non-empty: the top-of-stack integer is the index of the first byte of (the tag of) the effectively "top" mono record in the stack, the first byte of (the tag of) each non-top mono record in the stack immediately follows the last byte of (the value of) the immediately preceding mono record in the stack, and the last byte of (the value of) the effectively "bottom" mono record in the stack is the last byte of (the value of) the stack

record.The top mono record contained in a non-empty stack record is popped out by a microcode action which reads out (and then clears) the successive bytes of the top mono record while incrementing the top-of-stack integer, according to the tag of the top mono record. A new top mono record is pushed into a non-full stack record by a microcode action which decrements the top-of-stack integer while writing in the successive bytes of the new mono record in reverse order, according to the tag of the new mono record. A stack record is effectively full relative to an attempt to push in a new mono record if and only if the byte length of the new mono record exceeds the top-of-stack integer.

A vector poly record has two special subrecords: the length field (index -2), and the reference count field (index -1). The byte length of the space reserved within the vector record value for each non-special subrecord equals the byte length of the longest type of mono record, namely an instruction label which incorporates the sub-pointer form of <ip> and the pointer form of <ep>; each actual non-special mono subrecord of the vector record is left-justified in the space reserved for it.

A program-contour poly record has six special subrecords: the length field (index -6), the reference count field (index -5), the ordinal subrecord (index -4), the tag subrecord (index -3), the instruction-pointer subrecord (index -2), and the environment-pointer subrecord (index -1). The non-special subrecords of a program contour are handled in the same manner as for a vector record.

An execution-contour poly record has six special subrecords: the length field (index -6), the reference count field (index -5), the display management subrecord (index -4), the stack-pointer subrecord (index -3), the antecedent-pointer subrecord (index -2), and the environment-pointer subrecord (index -1). The non-special subrecords of a execution contour are handled in the same manner as for a vector record.

A virtual processor poly record has five special subrecords: the reference count field (index -5), the processor-identity register (index -4), the display-pointer register (index -3), the stack-pointer register (index -2), and the instruction-label register (index -1); the usual length field is absent since the CMA design provides 16 non-special subrecords for all virtual processors. The 16 non-special subrecords of a virtual processor are considered to be the 16 working registers R0, R1, ... , RE, RF, and they are handled in the same manner as for a vector record. A short field in the virtual processor tag serves to register under microcode control the dynamic state of the virtual processor. The processor identity register of a virtual processor contains a read-only pointer to the virtual processor itself; this register is set at the time of allocation of the virtual processor, and constitutes identification of the virtual processor.

The roles of the various special subrecords of poly records are further explained in secs. 2 and 3.

2. Nested Module Computation Static Snapshot Structure

A nested module computation is a time sequence of **snapshots**. Each snapshot is a self-contained data structure which has a natural decomposition into a **program component** and an **execution component**. A self-contained data structure is a data structure all of whose contained (sub)pointer mono record subrecords point (in)to poly records belonging to the data structure. The program component of a snapshot is a self-contained data structure; the execution component of a snapshot is generally not a self-contained data structure, since it usually contains (sub)pointers which point (in)to poly records of the program component. In this report, the program component of a computation is to be regarded as time-invariant, that is, the same in all snapshots of the computation. The execution component, however, evolves rapidly through successive snapshots of the computation. Thus a computation may quite properly be regarded as an execution of its program component, with the course of the execution being recorded in the execution component.

2.0 Static Structure of the Program Component

The program component is a self-contained data structure consisting of two parts: a **program skeleton**, and additional constant structure which is pointed to from within the program skeleton. The skeletal structure is of central importance for nested module computations and a specialized form and purpose, while the additional constant structure is specific to individual programs and can be quite arbitrary. Hence we describe in this report only the skeletal structure, leaving descriptions of the additional constant structure to considerations of specific programs.

The program skeleton is a forest structure composed of program contours and instruction records. The environment pointers of the program contours and instruction records constitute the links which realize the forest structuring of the program skeleton: each such environment pointer either is null or is a pointer to some program contour in the program skeleton. A record R in the program skeleton, either e program contour or an instruction record, is the root of some tree within the forest structure if and only if R is an instruction record. All non-null environment pointers in the program skeleton point away from tree leaves and toward tree roots. The instruction pointer of a program contour R in the program skeleton is a (sub)pointer which points (in)to some instruction record in the program skeleton whose environment pointer in turn is non-null and points to R. The following statements are consistent with all the above constraints and are presented to suggest the variety of program skeleton structures which can occur in practice; a program contour is never a tree leaf but may be a tree root; an instruction record is always a tree leaf, and may also be a tree root; an

instruction record need not be pointed (in)to by the instruction pointer of any program contour; and, if an instruction record I is pointed (in)to by the instruction pointer of some program contour R then the environment pointer of I is non-null and points to R.

2.1. Static Structure of the Execution Component

The execution environment is a generally non-self-contained data structure consisting of two parts: a dynamically varying **execution skeleton**, and additional dynamic structure which is is pointed to from within the execution skeleton. The dynamic skeleton is of central importance for nested module computations and has a specialized form and purpose, while the additional dynamic structure is specific to individual computations and can be quite arbitrary. Hence we describe in this report only the skeletal structure, leaving descriptions of additional structure to considerations of specific computations.

The execution skeleton is composed of two interconnected forest structures: the **access skeleton** and the **stack skeleton**.

2.1.0 The Access Skeleton

The access skeleton is a forest structure composed of execution contours and virtual processors. The environment pointers of the execution contours and the environment pointer portions of the instruction labels of the virtual processors constitute the links which realize the forest structuring of the access skeleton: each such environment pointer either null or is a pointer to some execution contour in the access skeleton. A record R of the access skeleton, either an execution contour or a virtual processor, is the root of some tree within the forest structure if and only if the environment pointer of R is null. Each virtual processor in the access skeleton is necessarily a leaf of some tree within the forest structure. All non-null environment pointers in the access skeleton point away from tree leaves and toward tree roots. An execution contour may be a tree leaf in the access skeleton; both execution contours and virtual processors may be tree roots in the access skeleton.

Within each snapshot, the access skeleton is related to the program skeleton as follows. The antecedent pointer of an execution contour in the access skeleton is a pointer which points to some program contour in the program skeleton. Let IP and EP be the instruction pointer and environment pointer portions of the instruction label of some virtual processor in the access skeleton; IP is necessarily a subpointer which points into some instruction record I of the program skeleton. EP is null if and only if the environment pointer of I is null. If EP is non-null then EP is necessarily a pointer which points to some execution contour EC in the access skeleton, and the antecedent pointer of EC is necessarily a pointer which points to some program contour PC in the program skeleton; the environment pointer of I is necessarily a pointer which points to PC.

2.1.1. The Stack Skeleton

The stack skeleton is a forest structure composed of stack records together with the execution contours and virtual processors comprising the access skeleton. The stack pointers of the stack skeleton records constitute the links which realize the forest structuring of the stack skeleton. A record R in the stack skeleton is the root of some tree within the forest structure if and only if the stack pointer of R is null. A record R in the stack skeleton is a leaf of some tree within the forest structure if and only if R is an execution contour or a virtual processor in the access skeleton. All non-null pointers in the stack skeleton point away from tree leaves and toward tree roots.

A stack record in the stack skeleton which is pointed to by the stack pointer of a virtual processor in the access skeleton is not pointed to by any other pointer in the execution component; the stack record is said to be attached to the virtual processor, and is used by that virtual processor for expression evaluation and for the bookkeeping associated with entries into and exits from program modules.

2.2. Environments

Within a snapshots of a nested module computation, three types of environments are of interest: access environments, program environments, and stack environments. An **access environment** is an environment within the **reduced access skeleton**, which is the forest structure comprising just the execution contours of the access skeleton. A **program environment** is an environment within the **reduced program skeleton**, which is the forest structure comprising just the program contours of the program skeleton. A **stack environment** is an environment within the **reduced stack skeleton**, which is the forest structure comprising just the stack records of the stack skeleton.

In any forest structure F whose links point away from tree leaves and toward tree roots, the concept of environment can be defined as follows. An environment E in F is any possibly empty sequence of elements of F, $E = \langle E(1),E(2),\ldots,E(n)\rangle$ with $n \geq 0$, which satisfies the following conditions: E(n) has a null link, and for $1 \leq i < n$ the link of E(i) points to E(i+1). The forest structure has exactly one empty environment: the empty sequence obtained by taking n=0. The first element E(1) and the last element E(n) of a non-empty environment

E in F are called the **top** and **bottom** elements of E, respectively. Environments within F are considered to be designated by mono records as follows. The mono record null designates the empty environment. If P is a pointer mono record which points to an element R of F, then P designates the environment within F whose top record is R.

2.2.0. Access Environments

Access environment are associated with label mono records and virtual processors as follows. Let L be either an instruction label or a contour label and let EP be the environment pointer of L. If EP is null, then the access environment associated with L is the empty environment. If EP is a pointer which points to an execution contour EC, then the access environment associated with L is the one whose top element is EC. If Π is a virtual processor having instruction label L, then the access environment associated with Π is the access environment associated with L.

2.2.1. Program Environments

Program environments are associated with label mono records, virtual processors, and access environments as follows. Let L be a contour label and let CP be the contour pointer of L; the program environment associated with L is the one designated by CP. Let L be an instruction label and let EP be the environment pointer of the instruction record (in)to which the instruction pointer of L necessarily points; the program environment associated with L is the one designated by EP. Let Π be a virtual processor having instruction label L; the program environment associated with Π is the one associated with L. Let E be an access environment. If E is empty, then the program environment associated with E is empty. If E is non-null and if AP is the antecedent pointer of the top execution contour in E, then the program environment associated with E is the one whose top element is the program contour which is necessarily pointed to be AP.

2.2.2. Stack Environments

Stack environments are associated with virtual processors as follows. Let Π be a virtual processor and let SP be the stack pointer of Π. If SP is null, then the stack environment associated with Π is empty. If SP is a pointer which points to a stack record SR, then the stack environment associated with Π is the one whose top stack record is SR.

Each virtual processor Π in the stack skeleton effectively has associated with it a single conceptual stack CS which can hold arbitrarily many mono records and whose sequence of contained mono records, in top down order, can be modified as follows. Let SE be the stack environment of Π. If SE is empty, then the sequence of mono records in CS is empty. But suppose that SE = <SR(1),SR(2),...,SR(n)>, with $n \geq 1$. The n the sequence of mono records in CS equals the concatenation of the sequence of mono records, in top down order, contained in the successive stack records SR(1), SR(2), ... , SR(n) of SE. While it is convenient to think in terms of the unbounded conceptual stack CS, in practice poly records must be allocated in fixed sizes so that the conceptual stack CS must be implemented as a linked stack of stack records.

2.3. Record Retention and Reference Counts

The contour model incorporates the following **principle of poly record retention**: a poly record R in the execution component of a computation must be retained - that is, not deallocated - if either R is an awake virtual processor or R is pointed to by (pointer portion of) a (sub)pointer which is a subrecord of some poly record other than R which must itself be retained.

The principle of poly record retention is upheld in CMA by the reference count mechanism. Each act of poly record allocation is performed by microcode on behalf of some awake virtual processor; the microcode allocates the poly record, constructs one pointer to the poly record, delivers the pointer to the virtual processor, and sets the reference count of the poly record to 1 to reflect the one pointer. Thereafter, each construction or destruction of a (sub)pointer whose pointer value points to the poly record is accompanied by an incrementation or decrementation of the reference count of the poly record, performed automatically by microcode. A poly record becomes a candidate for deallocation only when its reference count drops to zero. Three observations are in order. First, maintenance of reference counts requires implicit extra memory accesses. Second, not all deallocations allowed by the principle of retention are necessarily triggered by reference counts dropping to zero: garbage collection is generally also required. Third, memory allocation and deallocation mechanisms are beyond the scope of this report.

3. The Contour Model Assembly Language CMAL

The Contour Model Architecture CMA is somewhat unconventional in employing full tagging of both poly records and mono records and in automatically supporting the principle of retention. By contrast, CMA provides a rather conventional stack-oriented macro level programming language, the Contour Model Assembly Language CMAL. The purpose of this section is to detail the CMAL instruction repertoire, giving for many instructions before/after picture pairs to show graphically the net effects of instruction executions. While the

possibilities of underflow and overflow of virtual processor stacks must be checked for in the microcode realizations of many instructions, those possibilities are ignored in this report.

Throughout this section, let Π be the virtual processor executing the instruction specified, let I be the instruction record containing the instruction, and let S be the stack record pointed to by the stack pointer Π.sp of Π.

3.0. The CMA Instruction Cycle

As is usual, the CMA instruction cycle consists of three successive phases: instruction fetch, instruction pointer increment, and instruction execution. The instruction is fetched to an instruction register IR, which is necessarily part of the processing unit housing the virtual processor Π but is not considered part of Π itself since its content need never be saved with Π for later restoration when Π is removed from the processing unit. The bit lengths of individual machine instructions, although unspecified in this report, are intended to vary with the instruction in a manner partially dependent on the expected frequency of occurrence in normal coding; hence the amount by which the instruction pointer is incremented depends on the instruction being fetched. Except for this minor complication, the first two phases are straightforward; hence descriptions are given below for only the execution phases, the first two phases being assumed already completed.

3.1. Data Movement Between the Virtual Processor Stack and Registers

The virtual processor Π has 16 working registers, 5 special registers, and the stack S. The 16 working registers are denoted, in hexadecimal notation: R0 = Π.0, R1 = Π.1, ..., RE = A.E, and RF = Π.F. The 5 special registers are: Π.lab = Π.-1, Π.sp = Π.-2, Π.dp = Π.-3, Π.pid = Π.-4, Π.ref = Π.-5. The parts of Π.lab which hold the instruction subpointer and the environment pointer are regarded as pseudo registers, are denoted by Π.ip and Π.ep, are called the instruction pointer register and environment pointer register of Π, and may be accessed by using special instructions. The stack pointer Π.sp points to the stack S, the processor identity pointer Π.pid points to Π itself, the instruction pointer Π.ip points into the instruction record I immediately beyond the instruction just fetched, and the environment pointer Π.ep designates the current access environment of Π. The display pointer Π.dp either is null or is a pointer pointing to a vector poly record DV called the **display vector of Π**; when present, the display vector DV contains (sub)pointers pointing (in)to the execution contours in the access environment of Π which Π can utilize through special display instructions to gain access to the subrecords of those execution contours. All movements of mono records within the space composed of the 16 working registers, the 5 special registers, and the stack S either take place within S itself or take place between S and some set of registers; these cases are treated separately below.

3.1.0. Data Movement Within the Virtual Processor Stack

Execution of these instructions affects only mono records near the top of S.

---

00 nop — no-operation

Execution of this instruction has a vacuous effect.

---

01 pop

BEFORE AFTER

Execution of this instruction pops the top record from S.

---

02 dup — duplicate

BEFORE AFTER

Execution of this instruction duplicates the top record of S.

---

03 swp swap

BEFORE AFTER

Execution of this instruction interchanges the top two records of S.

04 cab

BEFORE AFTER

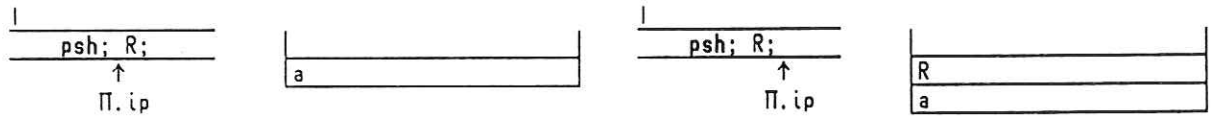
Execution of this instruction permutes the top three records of S so as to bring the third record to the top of S.

05 bca

BEFORE AFTER

Execution of this instruction permutes the top three records of S so as to place the top record below the next two records.

06 psh push constant



BEFORE AFTER

This instruction must be followed in I by some mono record R. Execution of this instruction pushes a copy of R onto S and increments Π.ip by an amount equal to the byte length of R.

3.1.1. Data Movement Between Working Registers and the Stack

The contents of working registers can be saved to the stack S or be restored from the stack S, either singly or in sets designated by register selectors. When a set of registers selected by a register selector M is saved to the stack, the selector M is pushed on top of the selected register contents to be available for controlling the later restoration of saved register contents to the same selected registers. Provision is also made for one virtual processor to cause the saving of a selected set of another virtual processor's working registers.

00 sav Rn save register Rn, 0≤n<16

BEFORE AFTER

Execution of the n-th one of these instructions, 0≤n<16, pushes onto S a copy of the content of register Rn.

01 res Rn — restore register Rn, 0≤n<16

BEFORE AFTER

Execution of the n-th one of these instructions, 0≤n<16, places into reister Rn a copy of the top record of S and pops that top record from S.

02 xch Rn — exchange register Rn, 0≤n<16

BEFORE AFTER

Execution of the n-th one of these instructions, 0≤n<16, exchanges the contents of register Rn and the top record of S.

03 sav — save selected registers

The top record of S must be a register selector M. Execution of this instruction comprises; first, M is popped from S to a micro register; second, for 16>i≥0 if M.i=1 then save Ri is effected; third, M is pushed back onto S.

BEFORE AFTER

A specific example of the execution of this instruction is illustrated above, with the 16-bit value of the register selector M expressed as four hexadecimal digits.

04 res — restore selected registers

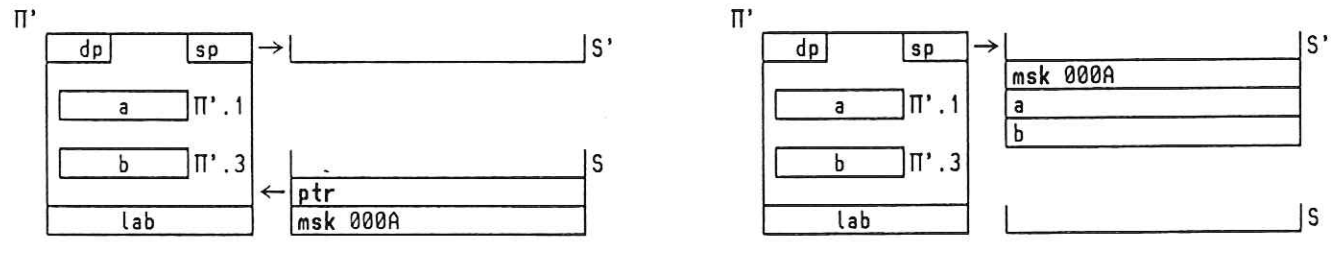
The top record of S must be a register selector M. Execution of this instruction comprises; first, M is popped from S to a microregister; second, for 16>i≥0 if M.i=1 then res Ri is effected. M is discarded rather than replaced onto S.

BEFORE AFTER

A specific example of the execution of this instruction is illustrated above, with the 16-bit value of the register selector M expressed as four hexadecimal digits.

05 tele save — save selected registers of another virtual processor

The top record of S must be a pointer P pointing to a virtual processor Π' which is asleep and which has a stack record S'; the second record of S must be a register selector M. Execution of this instruction comprises: first, P is popped from S to a microregister to give Π access to Π'; second, a copy of Π'.sp is placed into a microregister to give Π access to S'; third, M is popped from S to a microregister; fourth, for 16>i≥0 if M.i=1 then a copy of the content of the register Π'.i of Π' is pushed onto S'; fifth, M is pushed onto S'. The pointer P is discarded.



BEFORE AFTER

A specific example of the execution of this instruction is illustrated above, with the 16-bit value of the register selector expressed as four hexadecimal digits.

---

06 clr Rn — clear register Rn, 0≤n<16

Execution of the n-th one of these instructions, 0≤n<16, places null into Rn.

---

07 clr — clear selected registers

The top record of S must be a register selector M. Execution of this instruction comprises; first, M is popped from S to a micro register; second, for 16>i≥0 if M.i=1 then null is placed into register Ri. M is discarded.

BEFORE AFTER

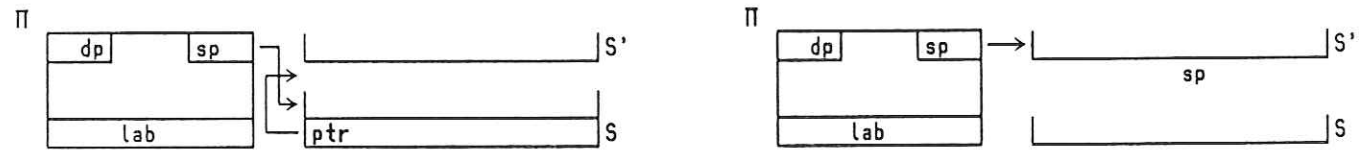
A specific example of the execution of this instruction is illustrated above, with the 16-bit value of the register selector M expressed as four hexadecimal digits.

3.1.2. Data Movement Between Special Registers and the Stack

The contents of certain special registers and pseodo registers can individually be saved to the stack S or restored from the stack S.

---

00 xch sp — exchange stack pointer

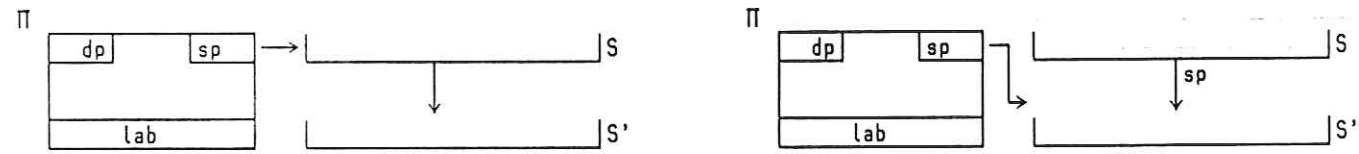


BEFORE AFTER

The top record of S must be a pointer P which points to a stack record S' whose reference count is 1. Execution of this instruction: copies the stack pointer Π.sp to the stack pointer subrecord S'.sp, and pops P from S to the stack pointer register Π.sp.

---

01 rev sp — revert stack pointer



BEFORE AFTER

The stack pointer subrecord S.sp must be a pointer P which points to a stack record S' whose reference count is 1. Execution of this instruction copies the stack pointer subrecord S.sp to the stack pointer register Π.sp.

---

02 xch dp — exchange display pointer

BEFORE AFTER

The top record of S must be a pointer P which points to a vector record DV whose reference count is 1. Execution of this instruction: pops P from S to a microregister, pushes a copy of the content of the display pointer register Π.dp onto S, and places the pointer P into the display pointer register Π.dp.

---

03 res dp — restore display pointer

BEFORE AFTER

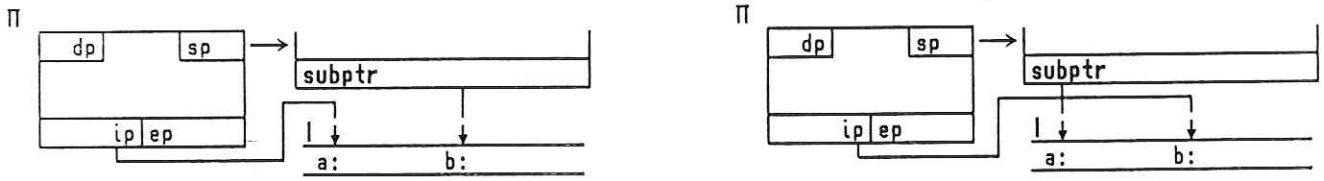
The top record of S must be a pointer P which points to a vector record DV whose reference count is 1. Execution of this instruction: pops the pointer P from S to the display pointer register Π.dp.

04 sav pid — save virtual processor identity

BEFORE AFTER

Execution of this instruction pushes onto S a copy of the content of the processor identity register Π.pid, namely a pointer to Π itself.

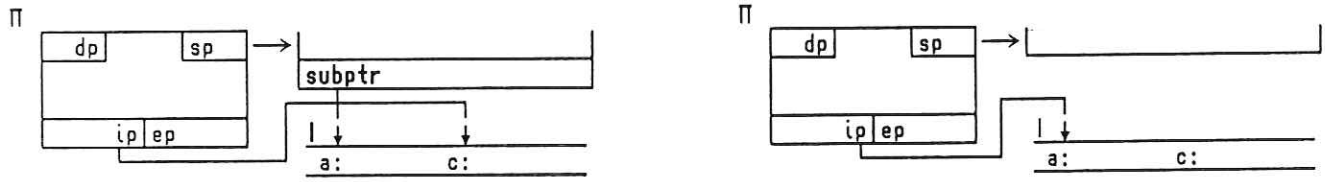
05 xch ip — exchange instruction pointer



BEFORE AFTER

The top record of S must be a subpointer P which points to an instruction within the current instruction record l; the instruction subpointer Π.ip, already incremented, also points to an instruction within l. Execution of this instruction exchanges the two subpointers P and Π.ip.

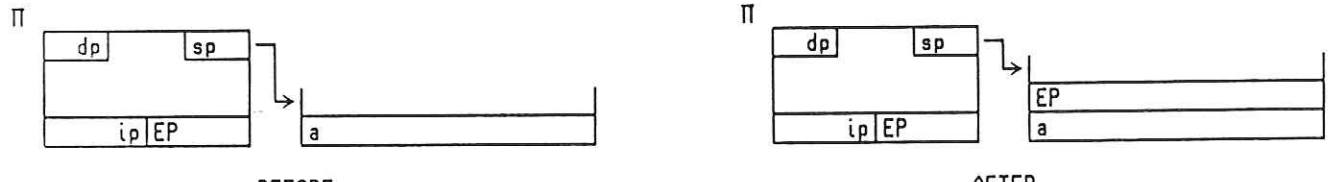
06 res ip — restore instruction pointer



BEFORE AFTER

The top record of S must be a subpointer P which points to an instruction within the current instruction record l. Execution of this instruction pops the subpointer P from S to the instruction pointer register Π.ip.

07 sav ep — save environment pointer



BEFORE AFTER

The top record of S must be a subpointer P which points to an instruction within the current instruction record l. Execution of this instruction pops the subpointer P from S to the instruction pointer register Π.ip.

3.2. Scaler Data Manipulation Instructions

We describe in this section some of the instructions which produce a single output scaler mono record from zero or more input mono records, both the input and output mono records being located at the top of the stack S. In executing one of these instructions on behalf of Π, the microcoded processing unit takes the following steps: first, an instruction-dependent number (the degree of the instruction) of input mono records are popped from S and placed into microregisters; second, a single output scaler mono record is produced from the input records and placed into a microregister; third, the output record is pushed onto the stack S. Normal execution of one of these instructions results in an output record having a tag specific to the instruction, and may require conditions on at least the tags if not the values of the input records as well. For simplicity in this report, we shall asume that all abnormal executions of these instructions result in the production of null as the output record. In the following subsections we specify the normal execution of each instruction by giving the normal tag for the output record, the degree of the instruction, any conditions required of input records, and a recipe for construction of the value, val U, of the output record U from the input records T (the top of stack record) and N (the next to top of stack record). The normal tag for the output record and the degree of the instruction are generally incorporated into the subsection headings.

3.2.0. Null Producing Instructions

3.2.0.0. Degree 0 Null Producing Instructions

---

00 mak null
This instruction is equivalent to the combination "psh; null".

3.2.1. Tag Producing Instructions

3.2.1.0. Degree 0 Tag Producing Instructions

---

00 mak tag mmm mmm ∈ <type mono>

BEFORE AFTER

Execution of one of these instructions pushes onto S an output record U whose tag is "tag" and whose value is the mono tag "mmm".

---

01 mak tag vp

BEFORE AFTER

Execution of one of these instructions pushes onto S an output record U whose tag is "tag" and whose value is the poly tag "vp" with the state bits set to "new".

---

02 imm mak tag ppp vp ≠ ppp ∈ <type poly>

BEFORE AFTER

This instruction must be followed in I by an integer mono record having a non-negative value "len". Execution of this instruction pushes onto S an output record U whose tag is "tag" and whose value is the poly tag "ppp int len".

3.2.1.1. Degree 1 Tag Producing Instructions

00 mak tag ppp vp ≠ ppp ε <type poly>

BEFORE AFTER

The top record T of S must be an integer mono record having a non-negative value "len". Execution of this instruction pops T from S and pushes onto S an output record U whose tag is "tag" and whose value is the poly tag "ppp int len".

01 tak tag

BEFORE AFTER

The top record T of S may be an arbitrary mono record. Execution of this instruction pops T from S and pushes onto S an output mono record U whose tag is "tag" and whose value is the tag "mmm" of T.

02 ind tak tag

BEFORE AFTER

The top record T of S must be either a pointer to a poly record R or a subpointer to mono subrecord (not a byte subrecord) R of some poly record. In either case, execution of this instruction pops T from S and pushes onto S an output record U whose tag is "tag" and whose value is the tag "rrr" of the mono or poly record R.

3.2.2. Logical Producing Instructions

3.2.2.0. Degree 0 Logical Producing Instructions

00 mak false
This instruction is equivalent to the combination "psh; log f".

01 mak true
This instruction is equivalent to the combination "psh; log t".

3.2.2.1. Degree 1 Logical Producing Instructions

00 not
T must be a logical mono record; val U = not (val T).

3.2.2.2. Degree 2 Logical Producing Instructions

00 xxx xxx ε {and,or,xor,nand,nor,...}
Both N and T must be logical mono records; val U = (val N) xxx (val T).

3.2.2.2.0. Degree 2 Arithmetic Relational Instructions

For each of these instructions both N and T must be integer mono records, while the normal output record is a logical mono record.

00 gt greater than
If (val N) > (val T) then U = log t ; otherwise, U = log f.

01 ge — greater than or equal to
If (val N) ≥ (val T) then U = log t ; otherwise, U = log f.

02 le — less than or equal to
If (val N) ≤ (val T) then U = log t ; otherwise, U = log f.

03 lt — less than
If (val N) < (val T) then U = log t ; otherwise, U = log f.

04 eq — equal to
If (val N) = (val T) then U = log t ; otherwise, U = log f.

05 ne — not equal to
If (val N) = (val T) then U = log f ; otherwise, U = log t.

3.2.2.2.1. Degree 2 General Relational Instructions

00 eq — equal to
N and T may be mono records of any types. If N and T are identical in both tag and value then U = log t ; otherwise, U = log f.

01 ne — not equal to
N and T may be mono records of any types. If N and T are identical in both tag and value then U = log f ; otherwise, U = log t.

02 ind eq — indirect equal to
Either N and T must both be pointer mono records pointing to poly records Q and S or else N and T must both be subpointer mono records pointing either to mono subrecords Q and S or to byte subrecords Q and S of arbitrary poly records. If Q and S are identical - as complete poly records , as mono records, or as single bytes - then U = log t ; otherwise, U = log f.

03 ind ne — indirect not equal to
Either N and T must both be pointer mono records pointing to poly records Q and S or else N and T must both be subpointer mono records pointing either to mono subrecords Q and S or to byte subrecords Q and S of arbitrary poly records. If Q and S are identical - as complete poly records , as mono records, or as single bytes - then U = log f ; otherwise, U = log t.

3.2.3. Register Selector Producing Instructions

3.2.3.0. Degree 0 Register Selector Producing Instructions

00 mask none — select no registers
This instruction is equivalent to the combination "**psh; msk 0000**".

01 mask all — select all registers
This instruction is equivalent to the combination "**psh; msk FFFF**".

3.2.3.1. Degree 1 Register Selector Producing Instructions

00 not — complement
T must be a register selector; val U = not (val T) (the 1's complement).

3.2.3.2. Degree 2 Register Selector Producing Instructions

00 xxx — xxx ε {and,or,xor,nand,nor,...}
N and T must both be register selectors; val U = (val N) xxx (val T) (each bit).

3.2.4. Integer Producing Instructions

3.2.4.0. Degree 0 Integer Producing Instructions

---

00 mak zero — make the integer 0
This instruction is equivalent to the combination "psh; int 0".

---

01 mak one — make the integer 1
This instruction is equivalent to the combination "psh; int 1".

---

02 tak stk len — take length of self stack record
The output record U is a copy of the integer length subrecord of the tag of S.

---

03 tak stk tos — take top of stack index of self stack record
Execution of this instruction pushes onto S an output record U which is a copy of the top of stack index of S as it was prior to the push.

3.2.4.1 Degree 1 Integer Producing Instructions

---

00 not — complement
T must be an integer mono record; val U = not (val T) (the 1's complement).

---

01 neg — negative
T must be an integer mono record; val U = - (val T).

---

02 abs — absolute value
T must be an integer mono record; val U = abs (val T).

---

03 tak len — take length of poly record
T must be a pointer mono record pointing to some poly record R. The output record U is a copy of the integer length subrecord of the tag of R unless R is a virtual processor, in which case the output record U is the integer mono record "int 16".

---

04 tak ref — take reference count of poly record
T must be a pointer mono record pointing to some poly record R. The output record U is a copy of the integer reference count subrecord of R.

---

05 tak tos — take top of stack index of stack record
T must be a pointer mono record pointing to some stack record R different from S. The output record U is a copy of the integer top of stack subrecord of R.

---

06 tele tak stk len — take length of tele stack record
T must be a pointer mono record pointing to some virtual processor Π' which is different from Π and has a stack record S'. The output record U is a copy of the integer length subrecord of the tag of S'.

---

07 tele tak stk tos — take top of stack index of tele stack record
T must be a pointer mono record pointing to some virtual processor Π' which is different from Π and has a stack record S'. The output record U is a copy of the integer top of stack subrecord of the tag of S'.

3.2.4.2. Degree 2 Integer Producing Instructions

For all of these instructions, both N and T must be integer mono records.

---

00 xxx — xxx ε {and,or,xor,nand,nor,...}
The values of N and T are treated as bits strings; val U = (val N) xxx (val T) (each bit).

---

01 add
val U = (val N) + (val T).

---

02 sub — subtract
val U = (val N) - (val T).

---

03 mpy — multiply

val U = (val N) x (val T).

04 intdiv — integer divide

This instruction is non-standard in that its normal execution produces two output records rather than just one. The value of T must be non-zero. Execution of this instruction pops both T and N from S to microregisters, pushes onto S a quotient integer mono record Q, and then pushes onto S a remainder integerrrr mono record R. The values of Q and R are uniquely determined by the following two conditions: $0 \leq$ (val R) < (abs (val T)) and val N = ((val T) x (val Q)) + (val R).

05 quot — integer division quotient

The value of T must be non-zero. This instruction is equivalent to the combination "**intdiv; pop**", and is a standard instruction producing only one output record.

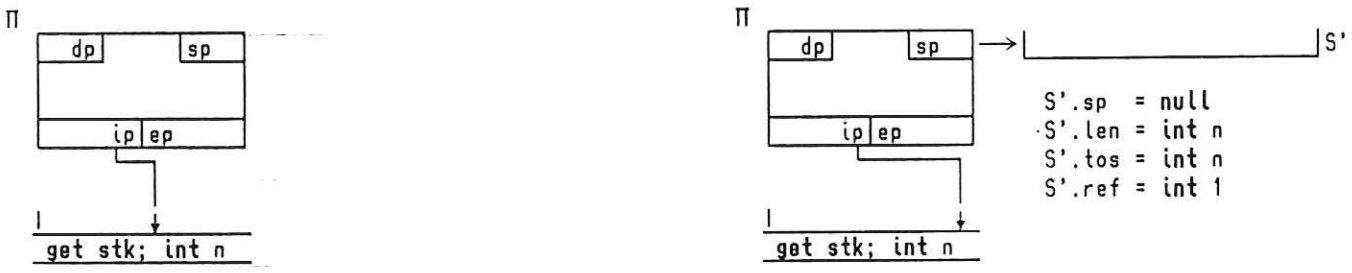
06 rem — integer division remainder

The value of T must be non-zero. This instruction is equivalent to the combination "**intdiv; swp; pop**", and is a standard instruction producing only one output record.

3.3. Record Allocation

Only poly records can be allocated; mono records occur only as subrecords of poly records. Existing poly records can be duplicated and new poly records can be freshly allocated. All poly record allocations produce records in the execution component. New poly records are initialized during allocation according to their types as follows. The state of a newly allocated virtual processor is **new**. The length of all virtual processors is 16, and hence the tag of a virtual processor does not incorporate a length subrecord. The length of a poly record which is not a virtual processor must be supplied as an operand for the allocate instruction, is incorporated into the tag of the poly record, and does not change during the lifetime of the poly record. Allocation of a poly record produces exactly one pointer to that poly record, and sets the reference count of the poly record to 1 to reflect the existence of this one pointer. Each character of a new text record is set to null. The ep and all code bytes of a new instruction record are set to null. The sp and all value bytes of a new stack record are set to null; the tos subrecord is set equal to the length subrecord to register the emptiness of the new stack record. All non-special subrecords of a new vector record are set to null. The ord, tag, ip, ep, and all non-special subrecords of a new program contour are set to null. The dsp, sp, ap, ep, and all non-special subrecords of a new execution contour are set tu null. The pid subrecord of a new virtual processor is set to be a pointe8.to the new virtual processor; this subrecord of the virtual processor does not change during the lifetime of the virtual processor and is never reflected in the reference count of the virtual processor. The dp, sp, lab, and all 16 working registers of a new virtual processor are set to null. The state of a virtual processor can be **awake** or **asleep** only if the label register of the virtual processor contains an instruction label.

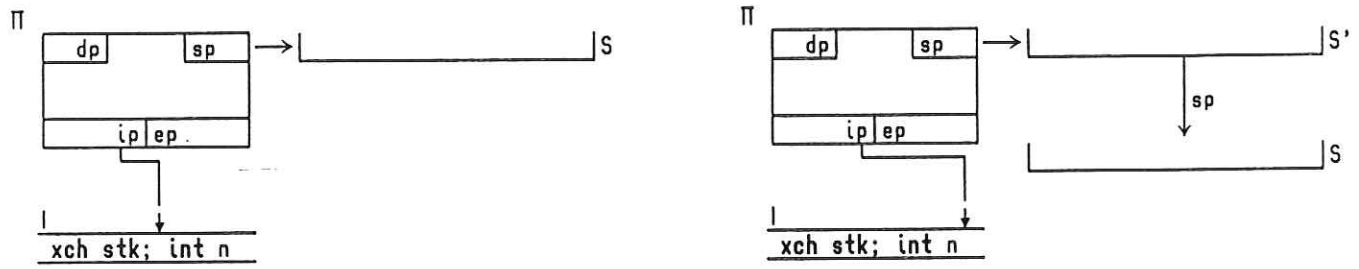
00 get stk — get a new stack record



This instruction must be followed in I by an integer mono record N = **int** n whose value part n is non-negative. The purpose of this instruction is to permit Π to acquire an empty new stack record S' whose length is the program constant n, regardless of whether or not Π already has a stack. Execution of this instruction fetches a copy of N from I using the instruction pointer Π.ip, increments Π.ip by an amount equal to the byte length of N, allocates a new stack record S' whose length subrecord is a copy of N, and places a pointer to S' into

the stack pointer register Π.sp.

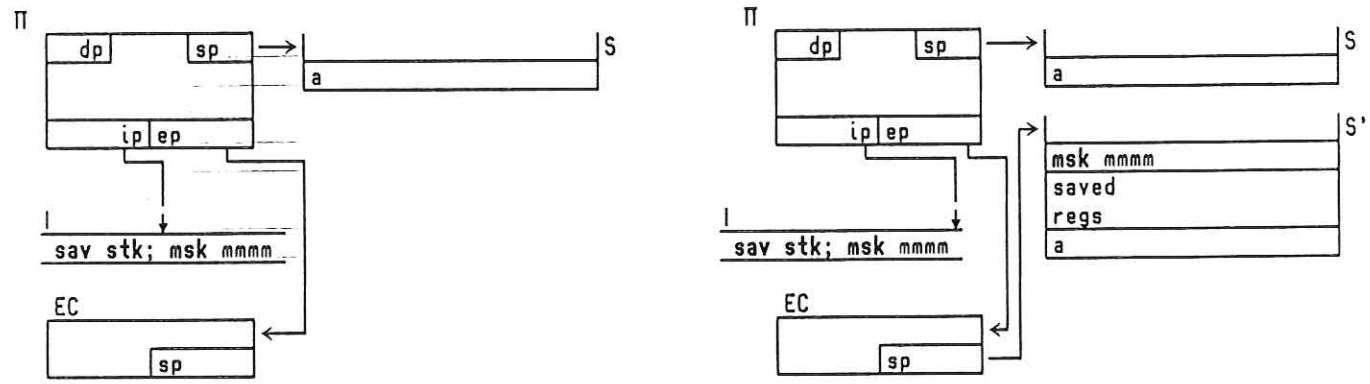
01 xch stk — exchange stack record



BEFORE AFTER

This instruction must be followed in I by an integer mono record N = int n whose value part n is non-negative. The purpose of this instruction is to permit Π, upon entry into a new program module, to exchange to a new empty stack record S' whose length is the program constant n, and which is automatically linked to the previous stack record S to permit later stack reversion back to S upon module exit. Execution of this instruction fetches a copy of N from I using the instruction pointer Π.ip, increments Π.ip by an amount equal to the byte length of N, allocates a new stack record S' whose length subrecord is a copy of N, places a copy of Π'.sp into the stack pointer subrecord S'.sp of S', and places a pointer to S' into the stack pointer register Π.sp.

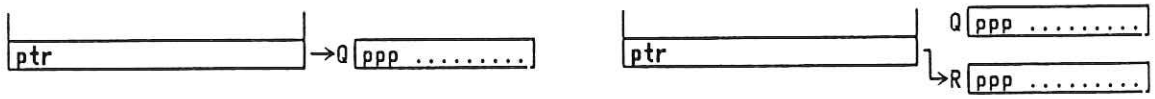
02 sav stk — save duplicate of stack record with selected registers



BEFORE AFTER

This instruction must be followed in I by a selector M = msk mmmm; the environment pointer Π.ep of Π must be a pointer pointing to some execution contour EC. The purpose of this instruction is to permit Π, upon entry into a contour module PC, to preserve via the stack pointer subrecord EC.sp for possible later use by a leap instruction an exact duplicate of S' of its stack record S together with copies of registers specified to be fixed during execution of PC. Execution of this instruction fetches a copy of M from I using the instruction pointer Π.ip, increments Π.ip by an amount equal to the byte length of M, allocates an exact duplicate S' of S, places a pointer to S' into EC.sp via the environment pointer Π.ep, performs relative to S' the register saving actions of sav, and pushes a copy of M onto S'.

03 alocopy — allocate copy of poly record



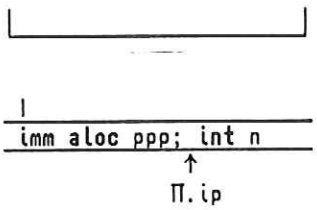
BEFORE AFTER

The top record T of S must be a pointer to some poly record Q. Execution of this instruction pops T from S to a microregister, allocates a poly record R which is an exact copy of the record Q with the two exceptions noted below, and pushes onto S a pointer to the duplicate record R. Exceptions: (1) the reference count of R is set to 1; (2) if Q is a virtual processor, then the state of R is set to new, R.pid is set to point to R rather than to Q, and both R.dp and R.sp are set to null.

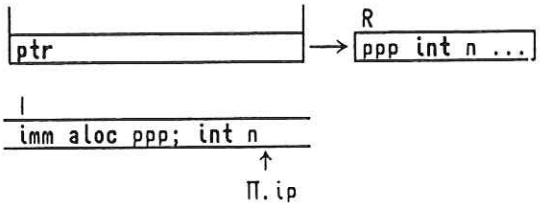
04 aloc vp — allocate virtual processor

Execution of this instruction allocates a virtual processor record Π', sets the state of Π' to **new**, sets the reference count of Π' to 1, sets Π'.pid to point to Π', sets all other registers of Π' to null, and pushes onto S a pointer to Π'.

05 imm aloc ppp — immediate allocate poly record, vp ≠ ppp ∈ <type poly>



BEFORE

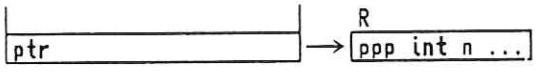


AFTER

This instruction must be follwed in I by an integer mono record N = **int** n whose value part n is non-negative. Execution of this instruction fetches a copy of N from I using the instruction pointer Π.ip, increments Π.ip by an amount equal to the byte length of N, allocates and properly initializes a new poly record R of type ppp whose length subrecord is a copy of N, and pushes onto S a pointer to the record R.

06 aloc ppp — allocate poly record, vp ≠ ppp ∈ <type poly>

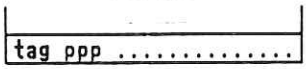
BEFORE



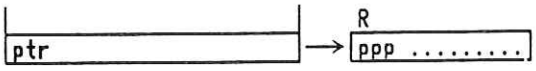
AFTER

The top record T of S must be an integer mono record T = int n whose value part n is non-negative. Execution of this instruction pops T from S to a micro register, allocates and properly initializes a new poly record R of type ppp whose length subrecord is a copy of T, and pushes onto S a pointer to the record R.

07 aloc — allocate poly record



BEFORE



AFTER

The top record T of S must be a tag record whose value part is a poly record tag P; if P is a virtual processor tag then the state of P is **new**. Execution of this instruction pops T from S to a micro register, allocates and properly initializes a new poly record R whose tag is a copy of the tag P, and pushes onto S a pointer to the record R.

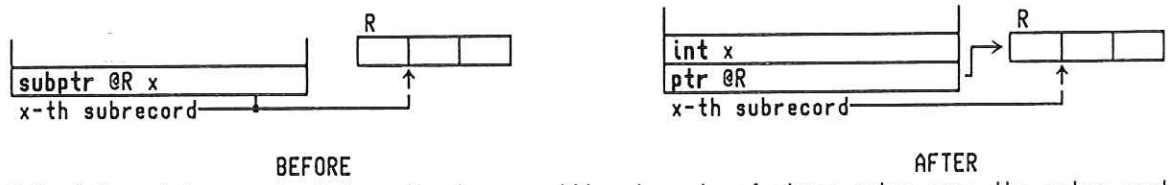
3.4. Pointer and Subpointer Operations: Selection and Index Modification

The instructions presented in this section capture most of the many address modification mechanisms found in contemporary architectures. Automatic indirection is purposely not incorporated into the memory reference instructions of CMAL; instead, intended access paths must be explicitly coded as CMAL instruction sequences.

The following types of operations involving pointers and subpointers can be performed using the instructions described in this section. A **break-subpointer** instruction breaks apart a subpointer into its two constituent parts: a pointer and an integer index. Six two-argument **select** instructions construct a subpointer from a pointer and an integer index. Six three-argument **modify-then-select** instructions construct a subpointer from a pointer and a register-resident integer index after applying an integer modification to the index, while six three-argument **select-then-modify** instructions construct a subpointer from a pointer and a register-resident integer index before applying an integer modification to the index. Three two-argument **modify-then-select** instructions save a copy of a register-resident subpointer to the stack S after applying an integer modification to the index portion of the subpointer, while three two-argument **select-then-modify** instructions save a copy of a register-resident subpointer to the stack S before applying an integer modification to the index portion of the subpointer. Finally, six two-argument **modify** instructions simply apply an integer modification to the index portion of a subpointer which may be in a register or on the stack S.

3.4.0. The Break-Subpointer Instruction

00 brk subptr — break subpointer



BEFORE AFTER

The top record T of S must be a subpointer, the two constituent parts of whose value are: the value part @R of a pointer P = ptr @R which points to some poly record R, and the value part x of an integer mono record X = int x; the subpointer T thus points to the x-th subrecord of R. Execution of this instruction pops T from S to a microregister, constructs from T and pushes onto S the pointer mono record P, and then constructs from T and pushes onto S the integer mono record X.

3.4.1. The Two-Argument Pointer Selection Instructions

Each of the six **select** instructions requires two input mono records: first, a pointer P = ptr @R to some poly record R; second, an integer index X = int x. The pointer P may be on the stack S or may be in some working register Ri, $0 \le i < 16$. The index X may be an immediate operand in the instruction, may be an the stack S, or may be in some working reister Rj, $0 \le j < 16$. Inputs which are on the stack S are denoted in the instruction by "*"; stack inputs are popped from the stack prior to formation of the output mono record. If X is an immediate operand then only its value x occurs within the instruction, coded as a short 2's-complement field. Each select instruction pushes onto the stack S as its single output mono record the subpointer Q = subptr @R x which points to the x-th subrecord of the poly record R.

00 sel *,x — P on stack, X immediate

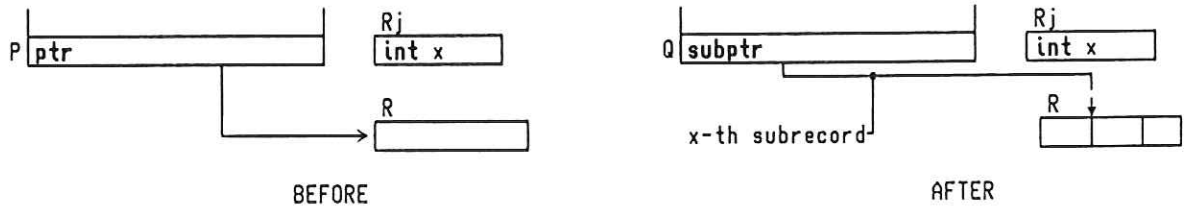
BEFORE AFTER

01 sel *,* — P on stack, X on stack

BEFORE AFTER

The pointer P must be below the index X, as shown; compare "**brk subptr**", 3.4.0.00.

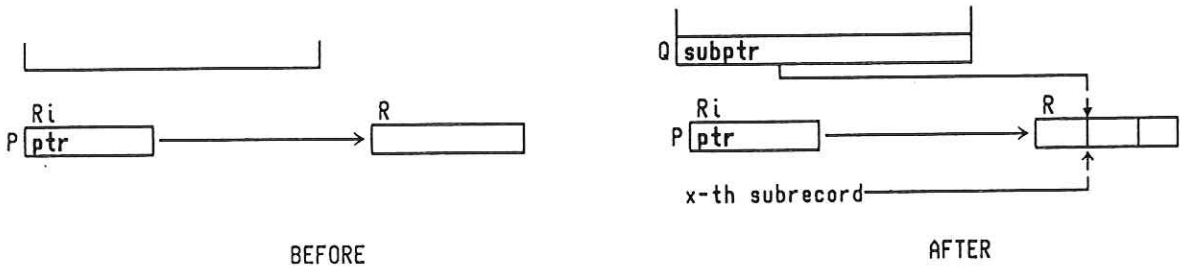
02 sel *,Rj — P on stack, X in Rj, $0 \le j < 16$



BEFORE AFTER

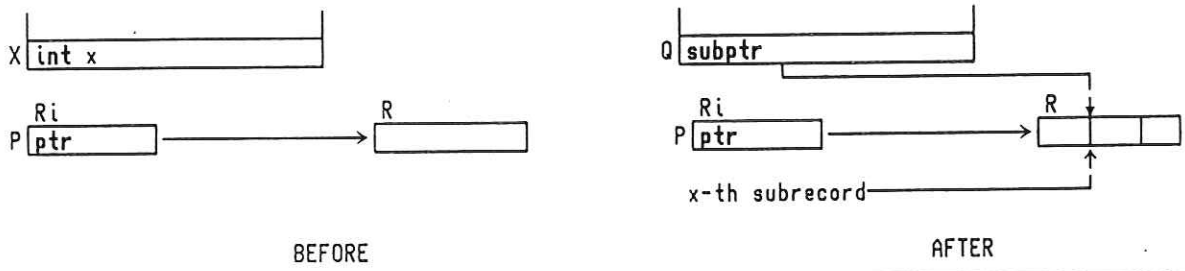
03 sel Ri,x

P in Ri, 0≤i<16, X immediate



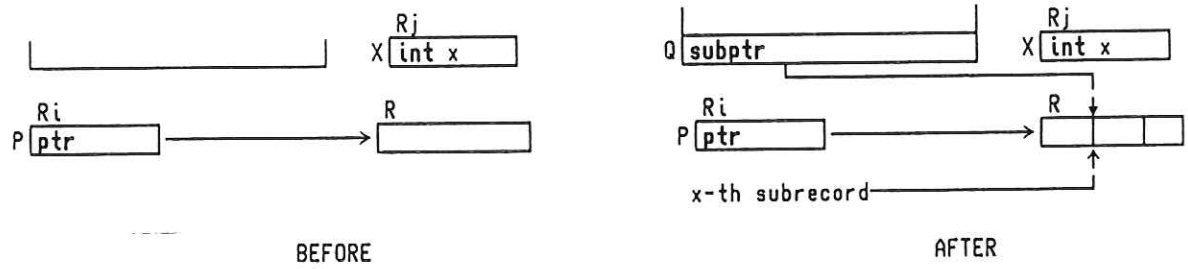
04 sel Ri,*

P in Ri, 0≤i<16, X on stack



05 sel Ri,Rj

P in Ri, X in Rj, 0≤i,j<16

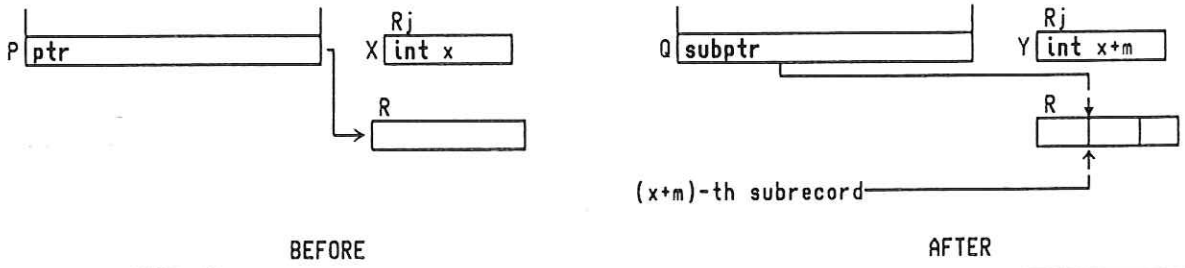


3.4.2. The Three-Argument Pointer Selection and Index Modification Instructions

Each of the six three-argument **modsel** (**modify-index-then-select-pointer**) instructions and each of the six three-argument **selmod** (select-pointer-then-modify-index) instructions requires three input mono records: first, a pointer P = ptr @R to some poly record R; second, a register-resident integer index X = int x; third, an integer index modifier M = int m. The pointer P may be on the stack S or may be in some working reister Ri, 0≤i<16. The integer index modifier M may be an immediate operand within the instruction, may be on the stack S, or may be in some working register Rk, 0≤k<16. Inputs which are on the stack S are denoted in the instruction by "*"; stack inputs are popped from the stack prior to formation of the output mono record. If M is an immediate operand then only its value m occurs within the instruction, coded as a short 2's complement field. Each modsel instruction first forms within the register Rj the modified index Y = int y, in which y = x+m, and then pushes onto the stack S as its single output mono record the subpointer Q = subpointer @R y which points to th y-th subrecord of the poly record R. Each selmod instruction first pushes onto the stack S as its sinle output mono record the subpointer Q = subptr @R x which points to the x-th subrecord of the poly record R, and then forms within the register Rj the modified index Y = int y, in which y = x+m.

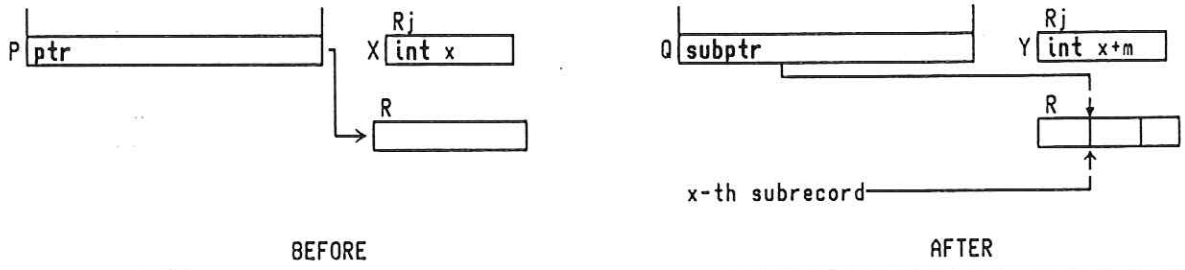
00 modsel *,Rj,m

P on stack, M immediate



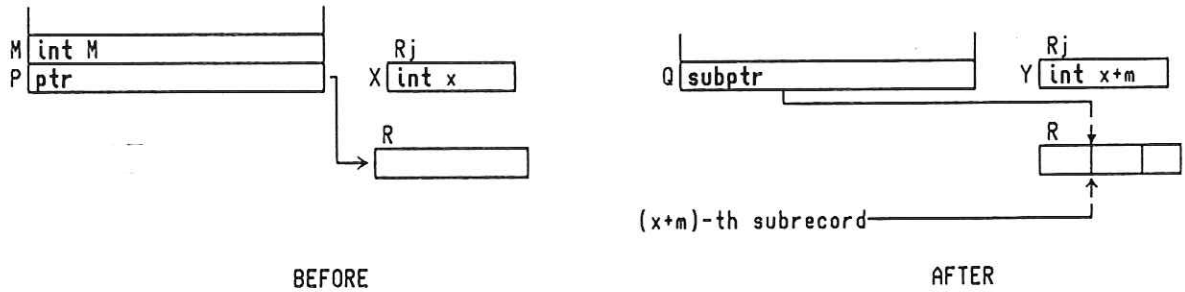
01 selmod *,Rj,m

P on stack, M immediate



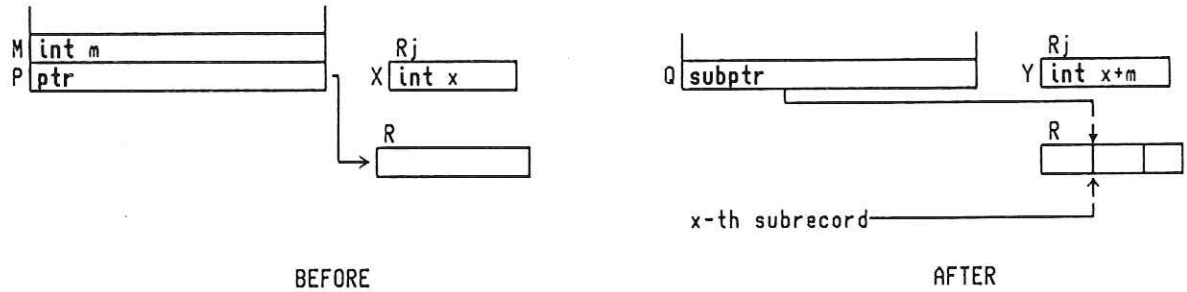
02 modsel *,Rj,*

P on stack, M on stack



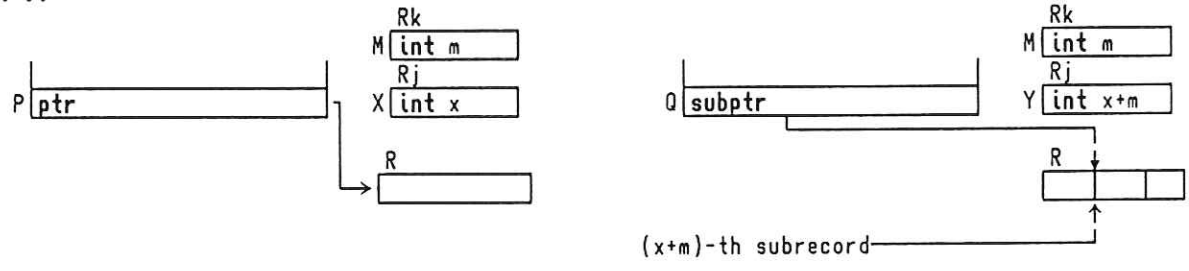
03 selmod *,Rj,*

P on stack, M on stack



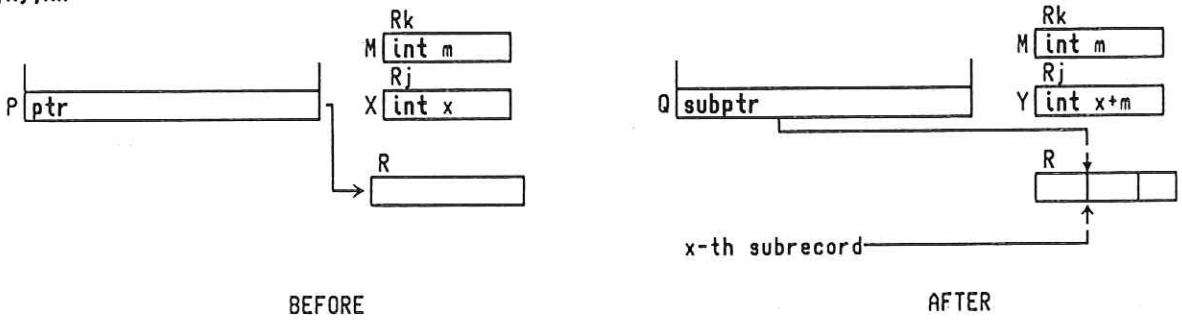
04 modsel *,Rj,Rk

P on stack, M in Rk, 0≤k<16



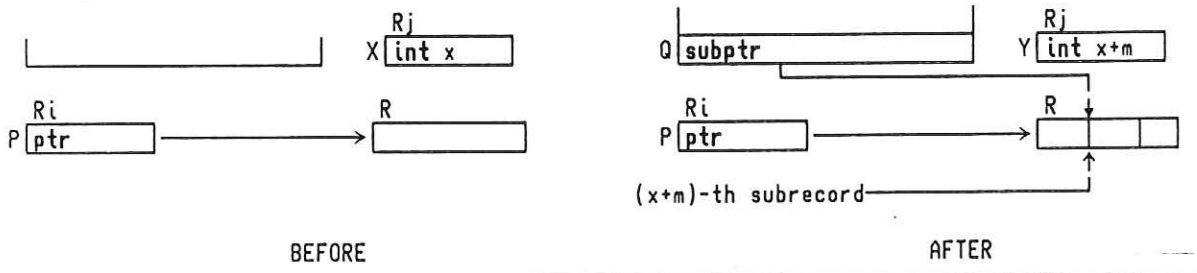
05 selmod *,Rj,Rk

P on stack, M in Rk, 0≤k<16



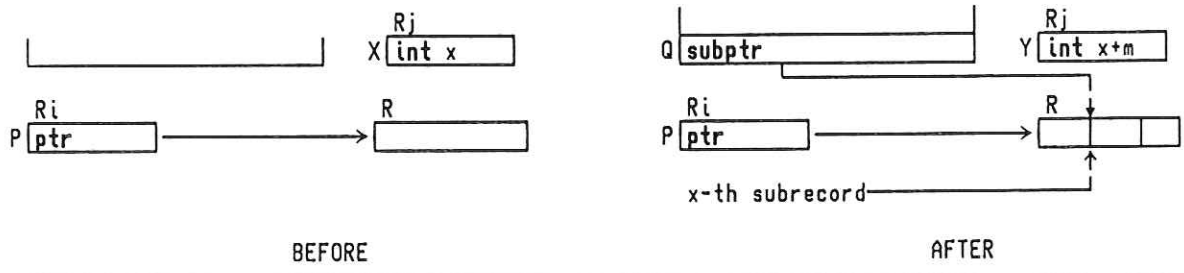
06 modsel Ri,Rj,m

P in Ri, 0≤i<16, M immediate



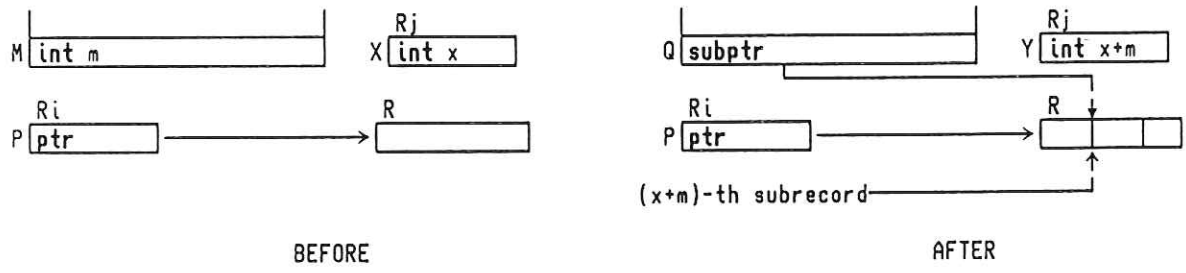
07 selmod Ri,Rj,m

P in Ri, 0≤i<16, M immediate



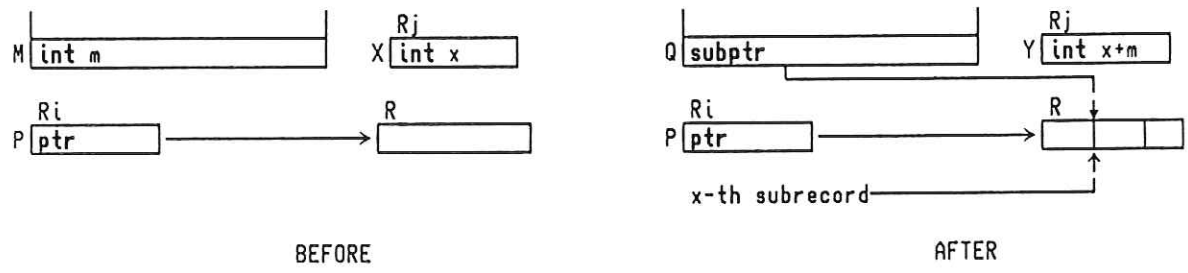
08 modsel Ri,Rj,*

P in Ri, 0≤i<16, M on stack



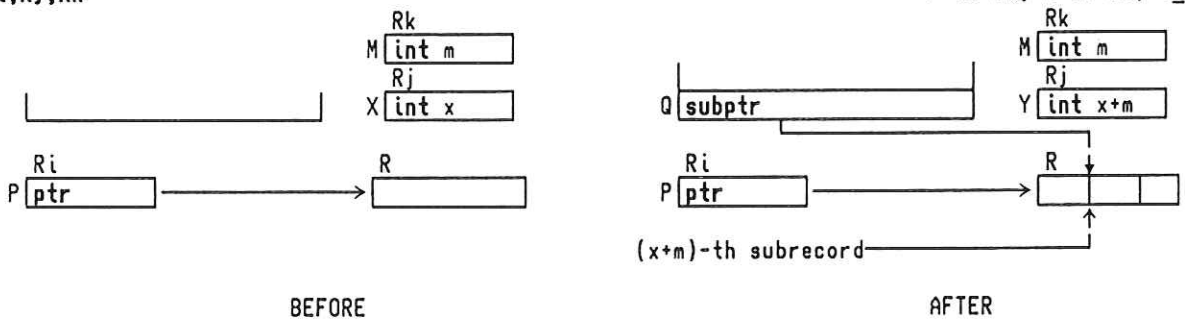
09 selmod Ri,Rj,*

P in Ri, 0≤i<16, M on stack



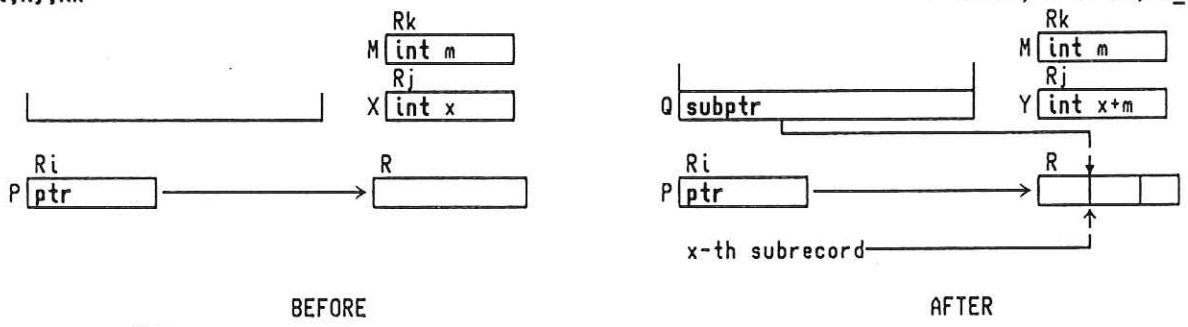
10 modsel Ri,Rj,Rk

P in Ri, M in Rk, 0≤i,k<16



11 selmod Ri,Rj,Rk

P in Ri, M in Rk, 0≤i,k<16

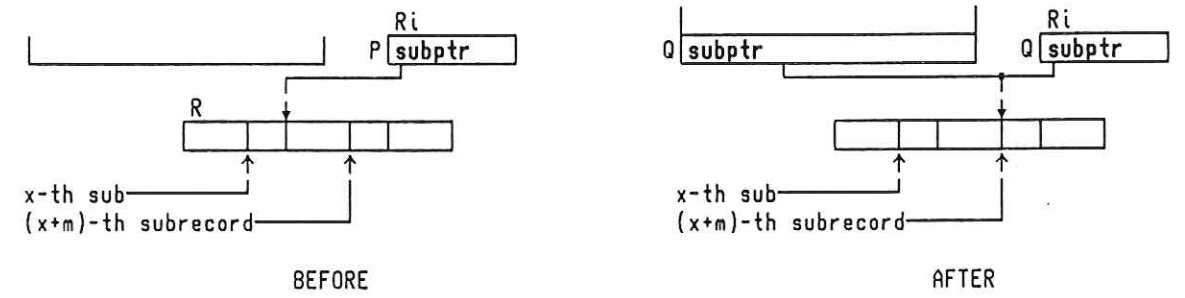


3.4.3. The Two-Argument Subpointer Index Modification and Selection Instructions

Each of the three two-argument **modsel** (modify-subpointer-index-then-**select**) instructions and each of the three two-argument **selmod** (**select**-then-modify-subpointer-index) instructions requires two input mono records: first, a register-resident subpointer P = subptr @R x which points to the x-th subrecord of some poly record R; second, an integer index modifier M = int m. The subpoiner P must be in some working register Ri, 0≤i<16. The integer index modifier M may be an immediate operand in the instruction, may be on the stack S, or may be in some workin register Rj, 0≤j<16. An input which is on the stack S is denoted in the instruction by "*"; a stack input is popped from the stack prior to formation of the output mono record. If M is an immediate operand then only its value m occurs within the instruction, coded as a short 2's complement field. ch modsel instruction forms within the register Ri the modified subpointer Q = subptr @R y, in which y = x+m and which points to the y-th subrecord of the poly record R, and then pushes onto the stack S as its single output mono record a copy of the subpointer Q. Each selmod instruction first pushes onto the stack S as its single output mono record a copy of the subpointer P, and then forms within the register Ri the modified subpointer Q = subptr @R y, in which y = x+m and which points to the y-th subrecord of the poly record R.

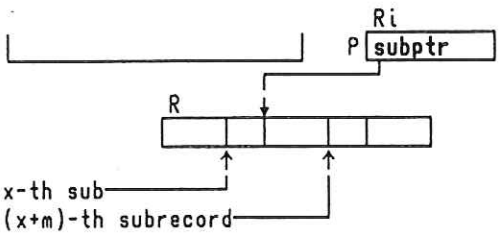
00 modsel Ri,m

M immediate

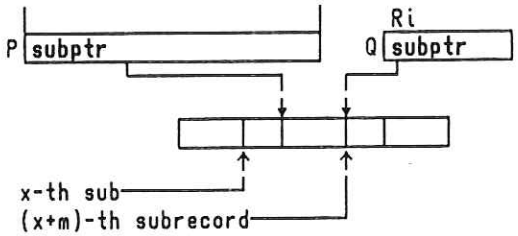


01 selmod Ri,m

M immediate



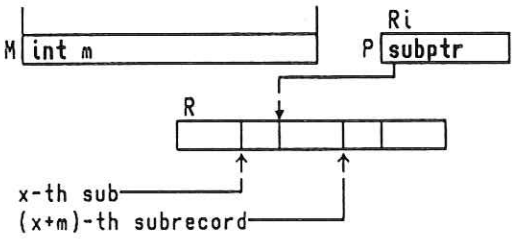
BEFORE



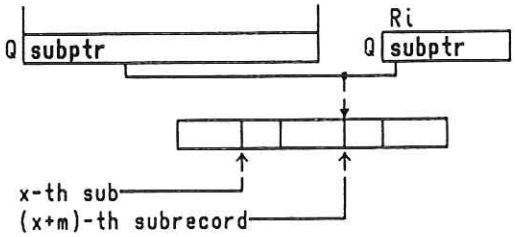
AFTER

02 modsel Ri,*

M on stack



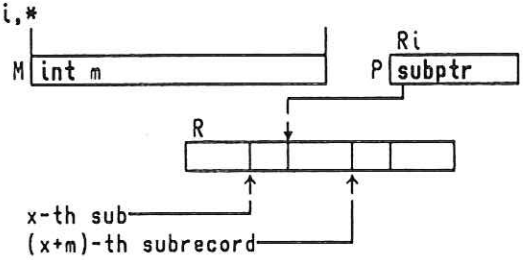
BEFORE



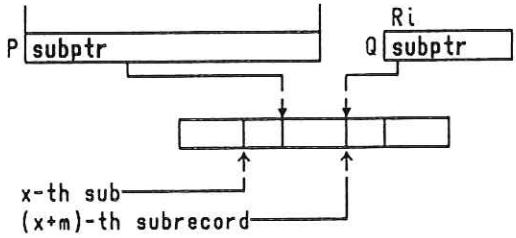
AFTER

03 selmod Ri,*

M on stack



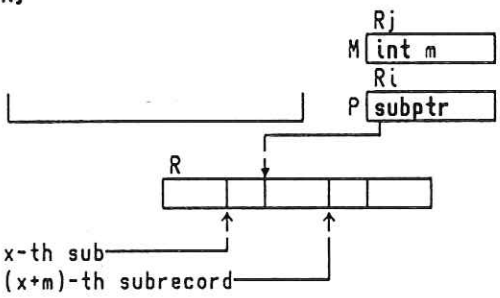
BEFORE



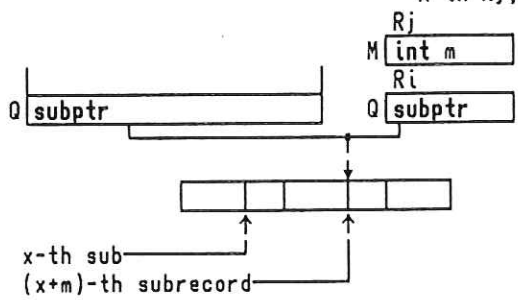
AFTER

04 modsel Ri,Rj

M in Rj, 0≤j<16



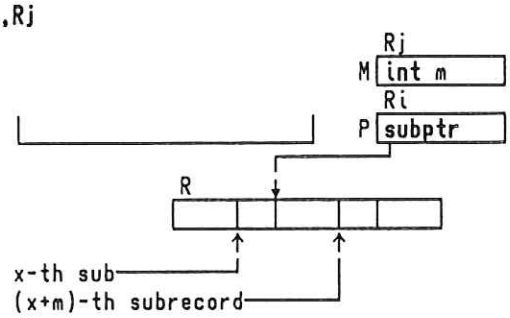
BEFORE



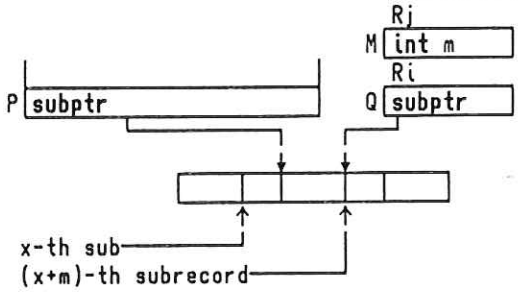
AFTER

05 selmod Ri,Rj

M in Rj, 0≤j<16



BEFORE



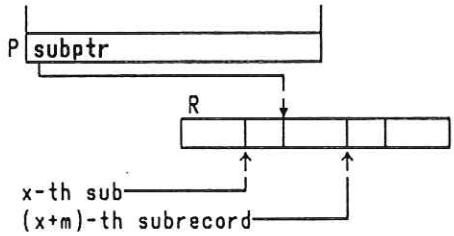
AFTER

3.4.4. The Two-Argument Subpointer Index Modification Instructions

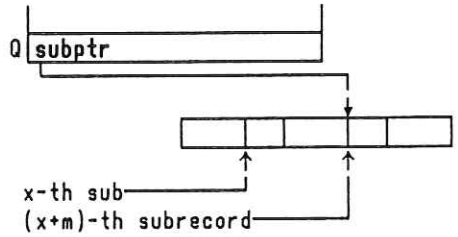
Each of the six two-argument **modsub** (modify-subpointer-index) instructions requires two input mono records: first, a subpointer P = subptr @R x which points to the x-th subrecord of some poly record R; second, an integer index modifier M = int m. The subpoiner P may be on the stack S or may be in some working register Ri, 0≤i<16. The integer index modifier M may be an immediate operand within the instruction, may be on the stack S, or may be in some workin register Rj, 0≤j<16. An input which is on the stack S is denoted in the instruction by "*"; stack inputs are popped from the stack prior to formation of the output mono record. If M is an immediate operand then only its value m occurs within the instruction, coded as a short 2's complement field. Each modsub instruction forms as its single output mono record the subpointer Q = subptr @R y, in which y = x+m and which points to the y-th subrecord of the poly record R. If P had been on the stack S then the output record Q is pushed onto S. If P had been in some working register Ri then the output record Q is placed into Ri.

00 modsub *,m

P on stack, M immediate



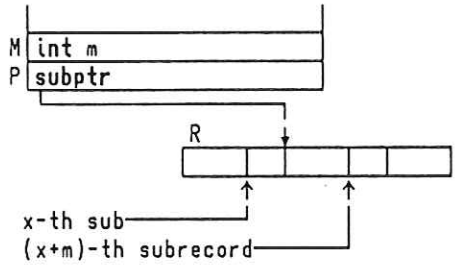
BEFORE



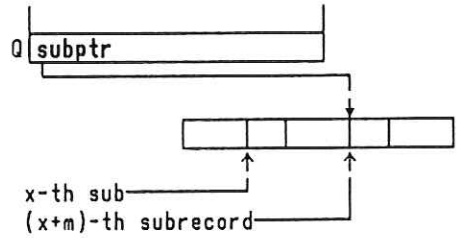
AFTER

01 modsub *,*

P on stack, M on stack

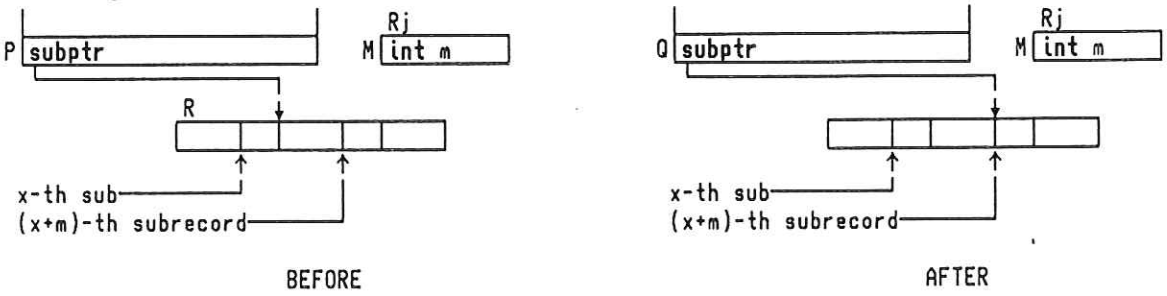


BEFORE

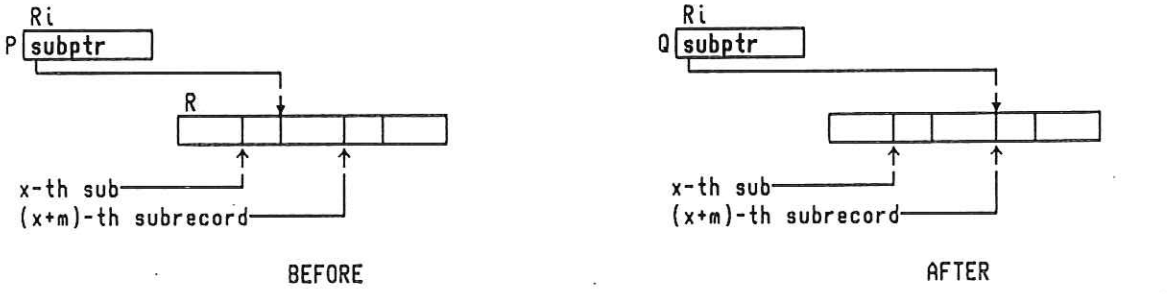


AFTER

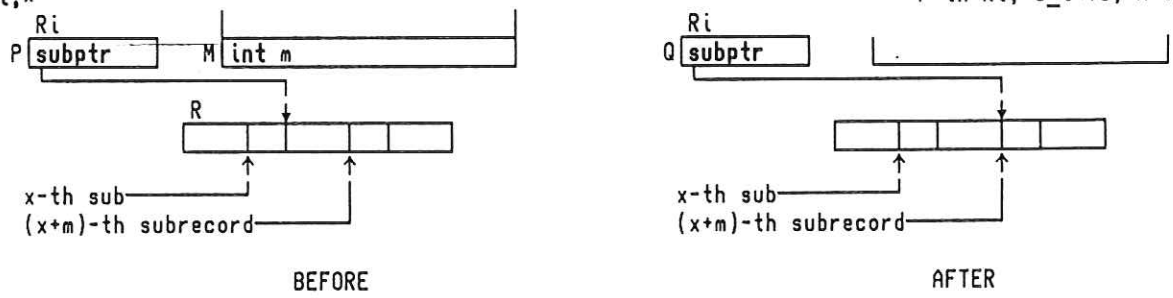
02 modsub *,Rj — P on stack, M in Rj, 0≤j<16



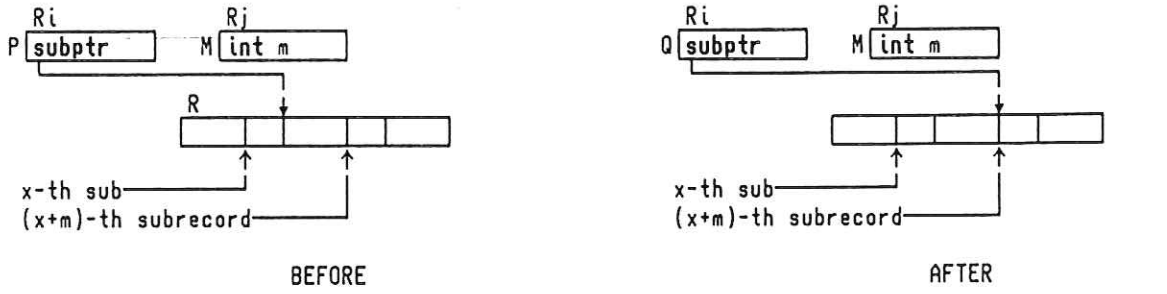
03 modsub Ri,m — P in Ri, 0≤i<16, M immediate



04 modsub Ri,* — P in Ri, 0≤i<16, M on stack



05 modsub Ri,Rj — P in Ri, M in Rj, 0≤i,j<16



3.5. Display Related Instructions

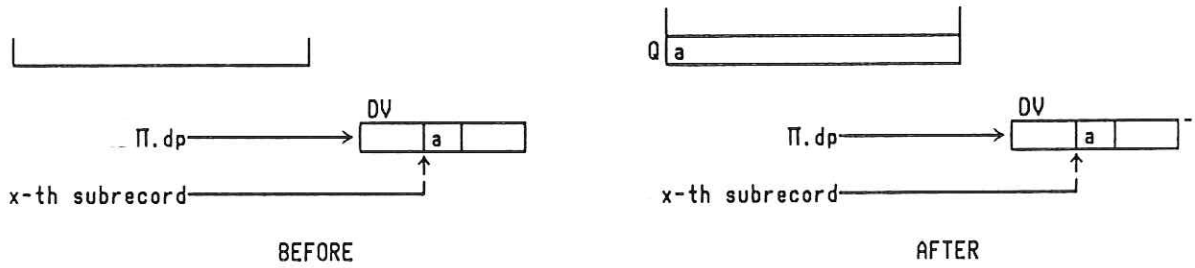
In this section we assume that the virtual processor Π has a display vector DV; that is, the display pointer register Π.dp contains a pointer P = ptr @DV pointing to the vector poly record DV. It is convenient to have specialized instructions both to maintain the display vector and to provide Π direct efficient use of the display vector in realizing access via identifiers. We describe below both : one-argument display instructions for fetching from and storing into the display vector DV, and two-argument display instructions which produce subpointers pointing into execution contours pointed to by subrecords of the display vector DV.

3.5.0. Display Vector Fetch Instructions

Each of the three one-argument **fet dsp** (fetch-from-display-vector) instructions requires one input mono record: an integer index X = int x. The integer index X may be an immediate operand in the instruction, may be on the stack S, or may be in some working register Ri, $0 \le i < 16$. An input which is on the stack S is denoted in the instruction by "*"; a stack input is popped from the stack prior to formation of the output mono record. If X is an immediate operand then only its value x occurs within the instruction, coded as a short 2's complement field. Each fet dsp instruction pushes onto the stack S aits single output mono record a copy Q of the x-th subrecord DV.x of the display vector DV.

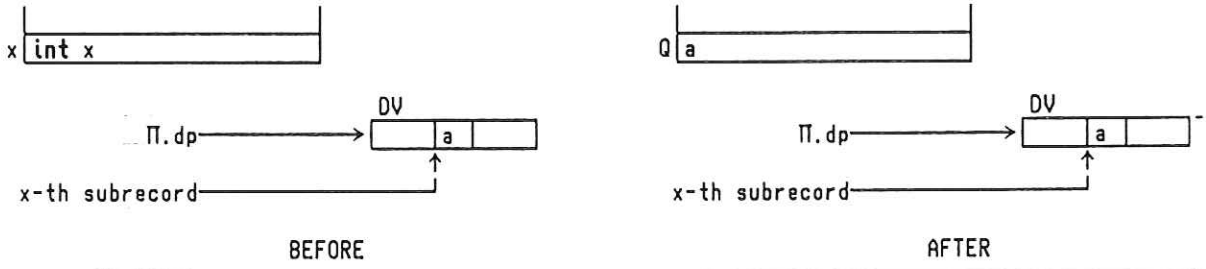
---

00 fet dsp x — x immediate



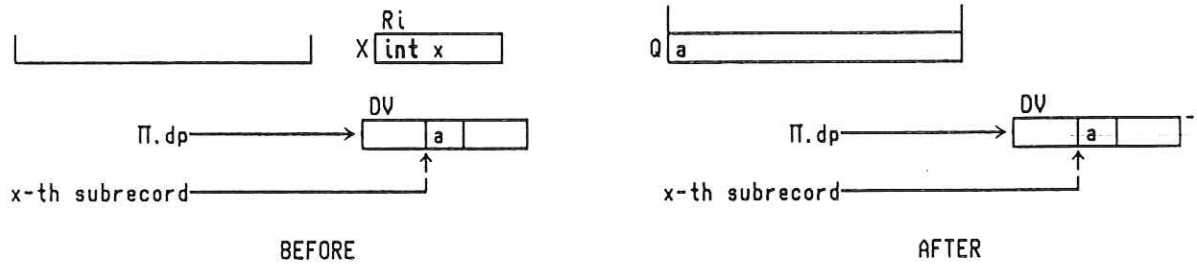
---

01 fet dsp * — x on stack



---

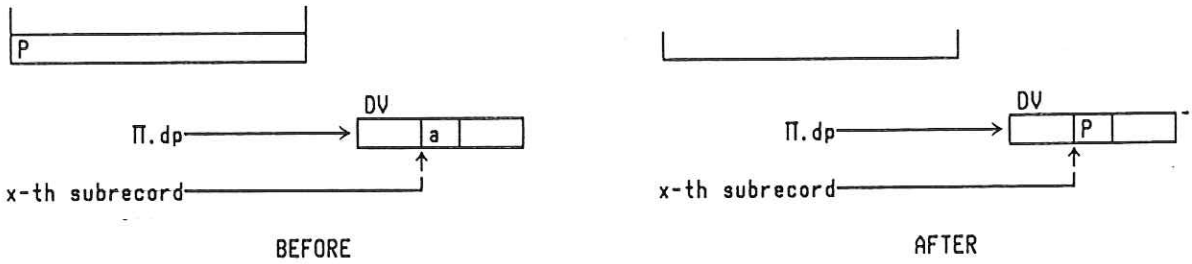
02 fet dsp Ri — x in Ri, $0 \le i < 16$



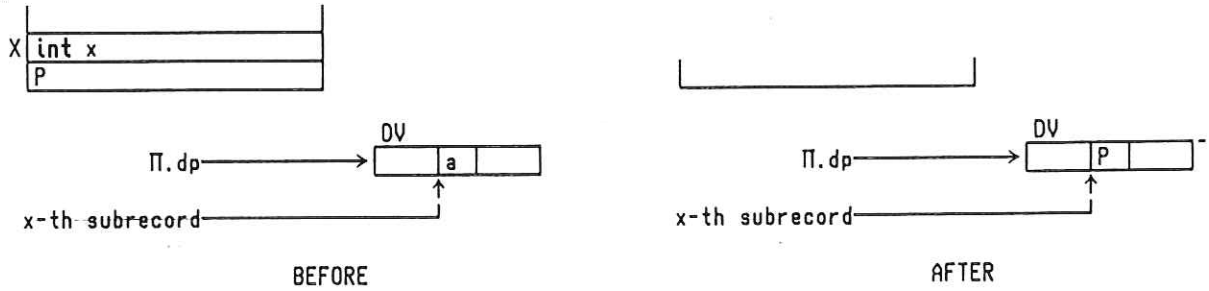
3.5.1. Display Vector Store Instructions

Each of the three one-argument **sto dsp** (store-into-display-vector) instructions requires two input mono records: first, a mono record P which is either null or is a (sub)pointer pointing (in)to some execution contour; second, an integer index X = int x. The mono record P must be on the stack S. The integer index X may be an immediate operand in the instruction, may be on the stack S, or may be in some working register Ri, $0 \le i < 16$. An input which is on the stack S is denoted in the instruction by "*"; a stack input is popped from the stack prior to formation of the output mono record. If X is an immediate operand then only its value x occurs within the instruction, coded as a short 2's complement field. Each sto dsp instruction stores the input record P into the x-th subrecord DV.x of th display vector DV.

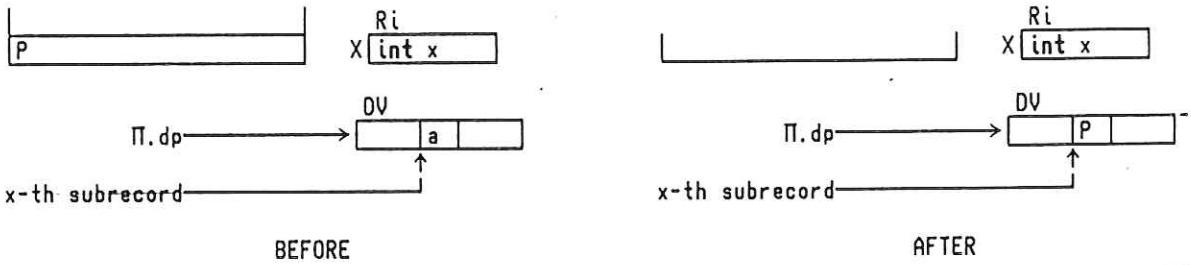
00 sto dsp x — x immediate



01 sto dsp * — x on stack



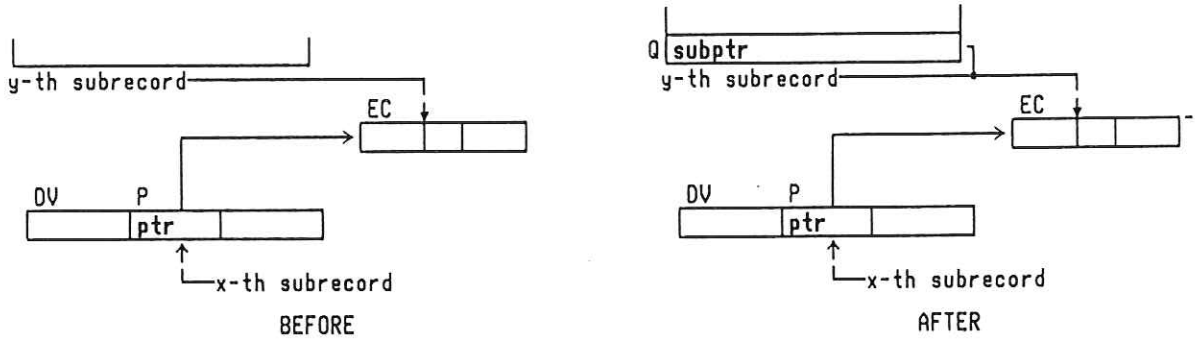
02 sto dsp Ri — x in Ri, 0≤i<16



3.5.2. Display Instructions for Accessing Into Execution Contours

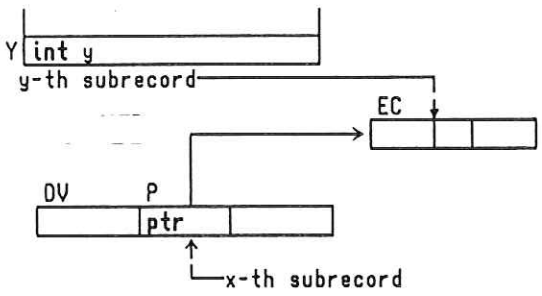
Each of the nine two-argument **dsp** (access-via-display) instructions requires two input mono records: first, an integer display vector index X = int x; second, an integer execution contour index Y = int y. The integer index X (resp. Y) may be an immediate operand in the instruction, may be on the stack S, or may be in some working register Ri, 0≤i<16 (resp. Rj, 0≤j<16). An input which is on the stack S is denoted in the instruction by "*"; a stack input is popped from the stack prior to the formation of the output mono record. If X (resp. Y) is an immediate operand then only its value x occurs within the instruction, coded as a short 2's complement field. The x-th subrecord of the display vector DV must be a pointer P = ptr @EC which points to some execution contour EC. Each dsp instruction fetches the pointer P from the display vector DV, forms from P the subpointer Q = subptr @EC y which points to the y-th subrecord of the execution contour EC, and pushes Q onto the stack S as its single output mono record.

00 dsp x,y — X immediate, Y immediate

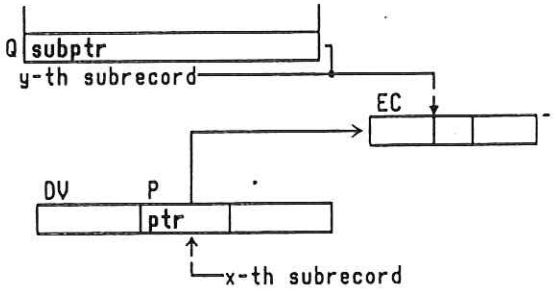


01 dsp x,*

X immediate, Y on stack



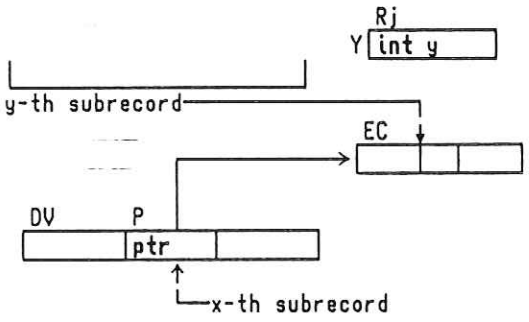
BEFORE



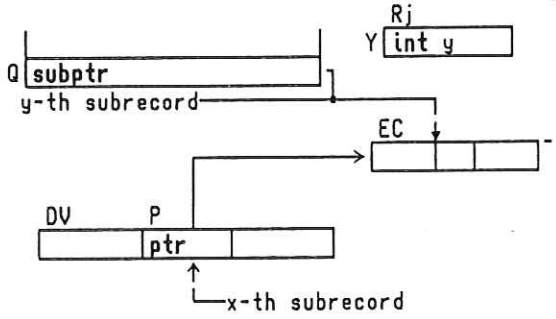
AFTER

02 dsp x,Rj

X immediate, Y in Rj, 0≤j<16



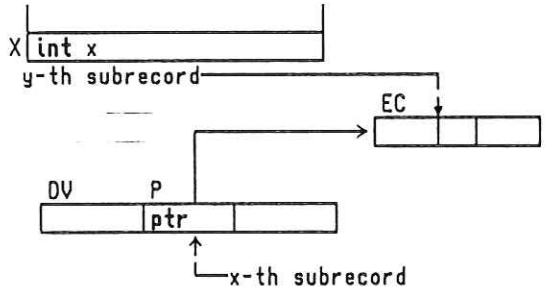
BEFORE



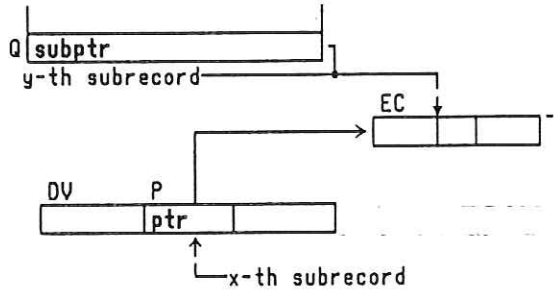
AFTER

03 dsp *,y

X on stack, Y immediate



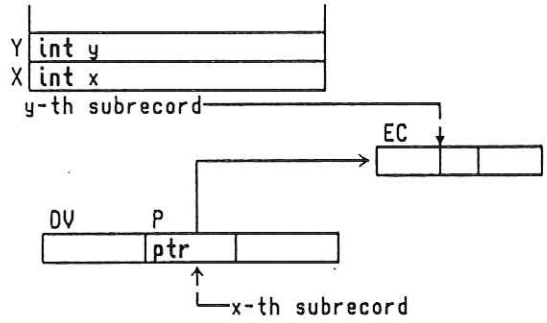
BEFORE



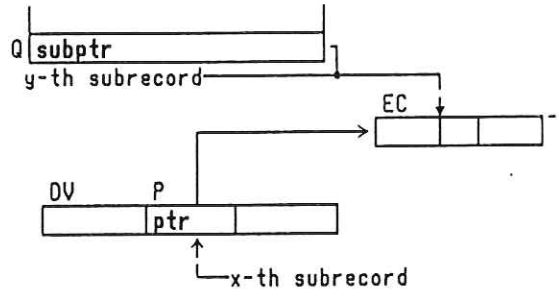
AFTER

04 dsp *,*

X on stack, Y on stack



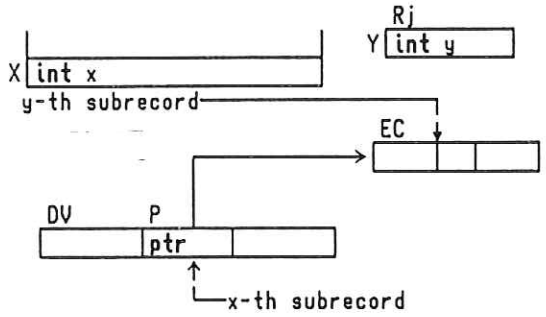
BEFORE



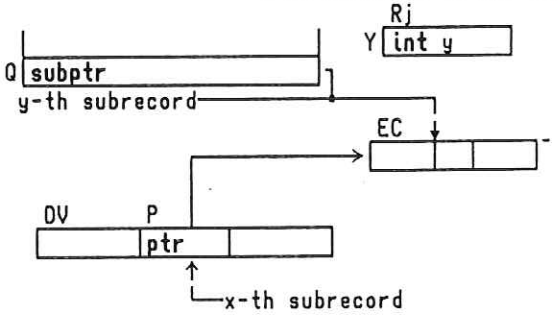
AFTER

05 dsp *,Rj

X on stack, Y in Rj, 0≤j<16



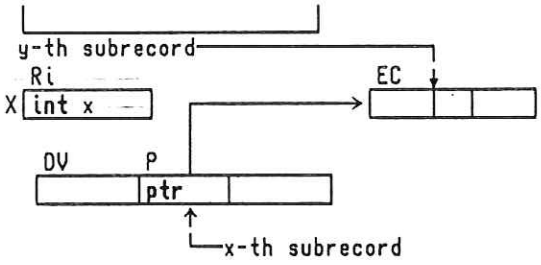
BEFORE



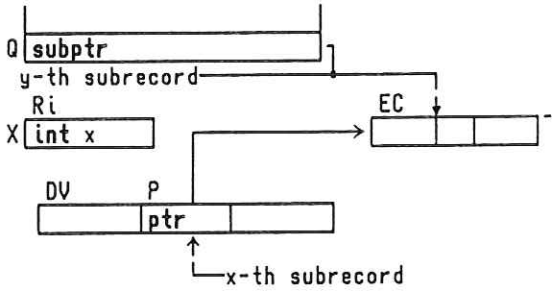
AFTER

06 dsp Ri,y

X in Ri, 0≤i<16, Y immediate



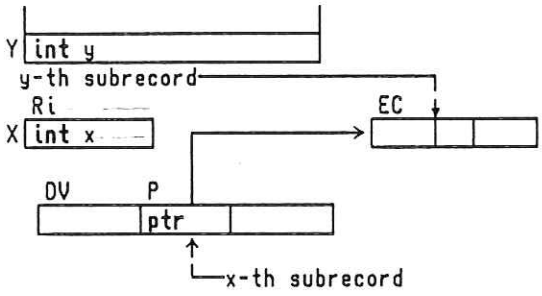
BEFORE



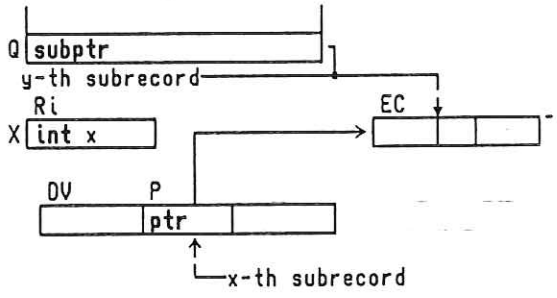
AFTER

07 dsp Ri,*

X in Ri, 0≤i<16, Y on stack



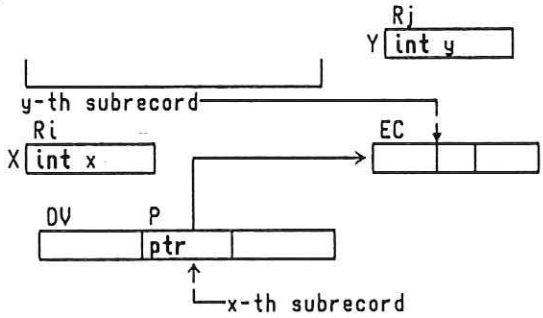
BEFORE



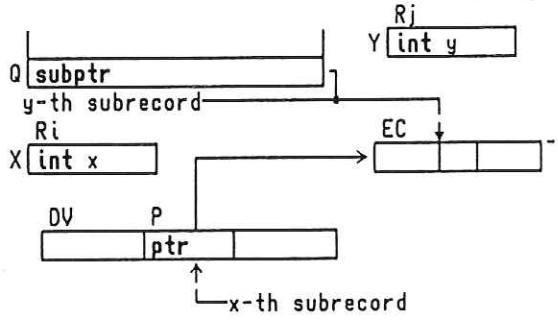
AFTER

08 dsp Ri,Rj

X in Ri, Y in Rj, 0≤i,j<16



BEFORE



AFTER

3.6. Label Manipulation Instructions

There are in CMA three types of label mono records: the instruction label (i-lab), the instruction procedure label (ip-lab), and the contour procedure label (cp-lab). The value of an instruction label IL is the concatenation of an instruction pointer IP and an environment pointer EP (((ee 1.0.26); IP must be a subpointer of non-negative index which points into some instruction record I having environment pointer subrecord I.ep, while EP must either be null or be a pointer which points to some execution contour EC whose antecedent pointer subrecord EC.ap necessarily points to some program contour PC; if EP is null then I.ep must be null, while if EP is non-null and points to EC then I.ep must be non-null and must point to PC. The value of an instruction procedure label IPL is the concatenation of an instruction pointer IP and an environment pointer EP (see 1.0.27); IP must be a pointer which points to some instruction record I having environment pointer subrecord I.ep, while EP must either be null or be a pointer which points to some execution contour EC whose antecedent pointer subrecord EC.ap necessarily points to some program contour PC; if EP is null then I.ep must be null, while if EP is non-null and points to EC then I.ep must be non-null and must point to PC. The value of a conour procedure label CPL is the concatenation of a contour pointer CP and an environment pointer EP (see 1.0.28); CP must be a pointer which points to some program contour PC, while EP must either be null or be a pointer which points to some execution contour EC; no other conditions need be met by CP and EP. The microcode realizations of those instructions described below which construct label mono records must check that the applicable conditions are met. CMAL contains both instructions which make labels from their constituent parts and instructions which break labels into their constituent parts.

---

00 brk i-lab — break instruction label

BEFORE AFTER

The top record of the stack S must be an instruction label IL whose value consists of an instruction pointer IP and an environment pointer EP. Execution of this instruction comprises: first, IL is popped from S to a microregister; second; a copy of EP is pushed onto S; third, a copy of IP is pushed onto S.

---

01 brk ip-lab — break instruction procedure label

BEFORE AFTER

The top record of the stack S must be an instruction procedure label IPL whose value consists of an instruction pointer IP and an environment pointer EP. Execution of this instruction comprises: first, IPL is popped from S to a microregister; second; a copy of EP is pushed onto S; third, a copy of IP is pushed onto S.

---

02 brk cp-lab — break contour procedure label

BEFORE AFTER

The top record of the stack S must be an contour procedure label CPL whose value consists of a contour pointer CP and an environment pointer EP. Execution of this instruction comprises: first, CPL is popped from S to a microregister; second; a copy of EP is pushed onto S; third, a copy of CP is pushed onto S.

---

03 mak i-lab — make instruction label

BEFORE AFTER

The top record of the stack S must be an instruction pointer IP which is a subpointer of non-negative index, while the next-to-top record of S must be an environment pointer EP. Execution of this instruction comprises: first, both IP and EP are popped from S to microregisters; second, checks are made to ensure that IP and EP satisfy the conditions described above; third, an instruction label IL whose value consists of copies of IP

and EP is formed and pushed onto S.

---

04 mak ip-lab — make instruction procedure label

BEFORE AFTER

The top record of the stack S must be an instruction pointer IP which is a pointer, while the next-to-top record of S must be an environment pointer EP. Execution of this instruction comprises: first, both IP and EP are popped from S to microregisters; second, checks are made to ensure that IP and EP satisfy the conditions described above; third, an instruction procedure label IPL whose value consists of copies of IP and EP is formed and pushed onto S.

---

05 mak cp-lab — make contour procedure label

BEFORE AFTER

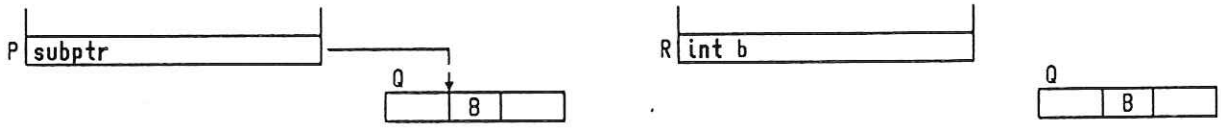
The top record of the stack S must be a contour pointer CP, while the next-to-top record of S must be an environment pointer EP. Execution of this instruction comprises: first, both CP and EP are popped from S to microregisters; second, checks are made to ensure that CP and EP satisfy the conditions described above; third, a contour procedure label CPL whose value consists of copies of CP and EP is formed and pushed onto S.

3.7. Data Movement Between the Virtual Processor and Memory

Only mono records can be moved between the virtual processor Π and memory. Each such movement of a mono record R occurs specifically between the stack S of Π and some poly record Q. The **fet** (fetch-to-stack) instruction provides for moving the mono record R from a source in the poly record Q and pushing R onto the stack S. The **sto** (store-from-stack) and **stor** (store-reverse-from-stack) instructions provide for popping the mono record R from the stack S and moving R to its destination within the poly record Q. The source or destination for the mono record R within the poly record Q must be designated either by a stack-resident pointer P which points to Q or by a stack-resident subpointer P which points into Q, as described below. The fetch instruction fet requires one stack-resident input mono record: as the top record of S, the (sub)pointer P. The store instruction sto requires two stack-resident input mono records: first, as the top record of S, the (sub)pointer P; second, as the next-to-top record of S, the mono record R. The store-reverse instruction stor requires two stack-resident input mono records: first, as the top record of S, the mono record R; second, as the next-to-top record of S, the the (sub)pointer P. Both the fetch and the store instructions pop their input mono records from the stack S to microregisters prior to completing their activities. When the poly record Q is a text, instruction, or stack record, special actions are required as described below.

---

00 fet — fetch and transform byte from text or instruction record

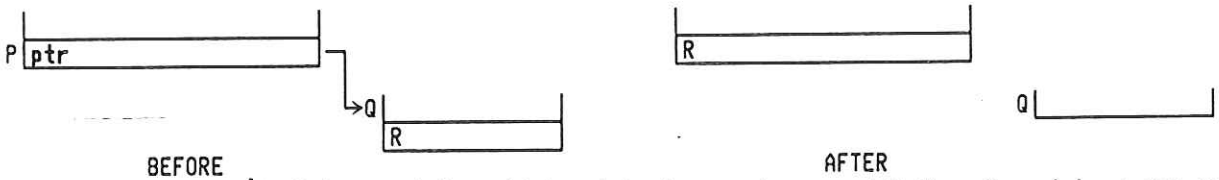


BEFORE AFTER

If the top record P of the stack S is a subpointer of non-negative index m which points into a text or instruction record Q then m must be less than the length of Q and P points to some text byte B within the value part of Q. Under these conditions, execution of this instruction comprises: first, P is popped from S to a microregister; second, an integer mono record R = int b, whose value contains a right-justified copy of B and is left-filled with 0 bits, is formed and pushed onto S.

---

00 fet — fetch mono record from stack record using pointer



BEFORE AFTER

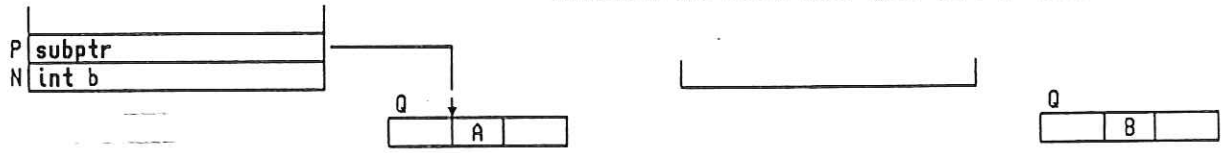
If the top record P of the stack S is a pointer which points to a poly record Q then Q must be a non-empty stack record which is not attached to any virtual processor. Under these conditions, execution of this instruction comprises: first, P is popped from S to a microregister; second, the top mono record R of the stack record Q is popped from Q and pushed onto S.

00 fet — fetch mono record using subpointer

BEFORE AFTER

If the top record P of the stack S is a subpointer of index m which points into a poly record Q, and if it is not the case both that m≥0 and that Q is either a text record or an instruction record, then the following conditions must hold: if m<0 then m is the index of some special subrecord of Q; if m≥0 then m is less than the length of Q and Q is not a stack record; if Q is a virtual processor then Q is not awake and P does not point at either the display pointer register of Q or the stack pointer register of Q. Under these conditions, the subpointer P points to some subrecord R of Q and execution of this instruction comprises: first, P is popped from S to a microregister; second, a copy of R is pushed onto S.

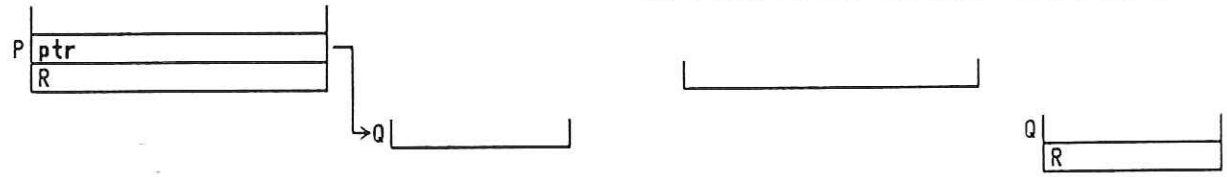
01 sto — transform and store byte into text or instruction record



BEFORE AFTER

If the top record P of the stack S is a subpointer of non-negative index m which points into a text or instruction record Q then the following conditions must hold: Q is in the execution component; m is less than the length of Q; P points to some byte C in the value part of Q; and the next-to-top record in S must be an integer mono record N = int b. Under these conditions, execution of this instruction comprises: first, both P and N are popped from S to microregisters; second, the byte C within Q is overwritten with a byte B which is a copy of the rightmost eight bits of the value b of N.

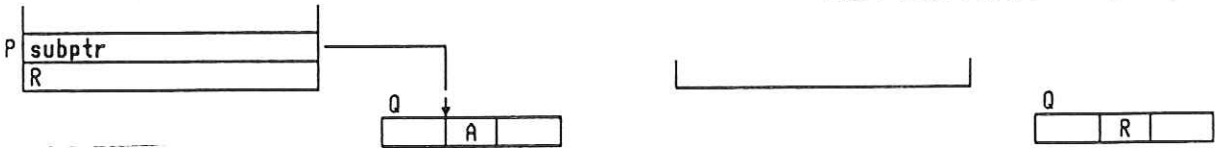
01 sto — store mono record into stack record using pointer



BEFORE AFTER

If the top record P of the stack S is a pointer which points to a poly record Q then the following conditions must hold: Q is in the execution component; Q is a stack record which is not attached to any virtual processor; the next-to-top record in S can be an arbitrary mono record R; and Q is sufficiently non-full that it can accept a copy of R. Under these conditions, execution of this instruction comprises: first, both P and R are popped from S to microregisters; second, a copy of R is pushed onto Q.

01 sto store mono record using subpointer



BEFORE AFTER

If the top record P of the stack S is a subpointer of index m which points into a poly record Q, and that it is not the case both that m≥0 and that Q is either a text record or an instruction record, then the following conditions must hold: Q is in the execution component; if m<0 then m is the index of some special subrecord of Q which is writeable (that is: not the length subrecord, not the reference subrecord, not the tos subrecord of a stack record, and not the pid subrecord of a virtual processor); if m≥0 then m is less than the length of Q and Q is not a stack record; if Q is a virtual processor then Q is not awake; and the next-to-top record in S can be an arbitrary mono record R, except that only an instruction label can be stored into the label reister of a virtual processor. Under these conditions, execution of this instruction comprises: first, P and R are popped from S to microregisters; second, a copy of R is stored into the m-th subrecord of Q.

02 stor reverse store

This instruction is equivalent to the combination "**swp; sto**".

3.8. Control Instructions

Control instructions are used to realize the following types of control activities: branches from within an instruction record to points within the same instruction record; leap's (goto's) to sites of activity determined by instruction labels; module entry and exit; and virtual processor state modification.

3.8.0. Branches

The CMAL representation of the instruction stream portion of an instruction record comprises a sequence of CMAL instructions optionally labelled by identifier-colon pairs. The five instructions defined below provide both for conditional and unconditional branching and for subroutine calling, to points within the current instruction record which are designated either statically by identifier-colon label pairs or dynamically by previosly obtained instruction labels.

00 b @L branch unconditionally to the instruction labelled L

There must be a unique instrution T within I which is labelled L. Execution of this instruction causes the instruction pointer of Π to be changed unconditionally to point to the target instruction T.

01 bt @L branch if true to the instruction labelled L

There must be a unique instrution T within I which is labelled L, and the top record R of the stack S must be a logical mono record. Execution of this instruction pops R from S to a microregister; if R is log t then the instruction pointer of Π is changed to point to T; if R is log f then the instruction pointer of Π is left pointing to the instruction immediately following the branch.

02 bf @L branch if false to the instruction labelled L

There must be a unique instrution T within I which is labelled L, and the top record R of the stack S must be a logical mono record. Execution of this instruction pops R from S to a microregister; if R is log f then the instruction pointer of Π is changed to point to T; if R is log t then the instruction pointer of Π is left pointing to the instruction immediately following the branch.

03 jmp unconditional jump

This instruction is identical to the "res ip" instruction defined in 3.1.2.06.

04 jsb unconditional subroutine call

This instruction is identical to the "xch ip" instruction defined in 3.1.2.05.

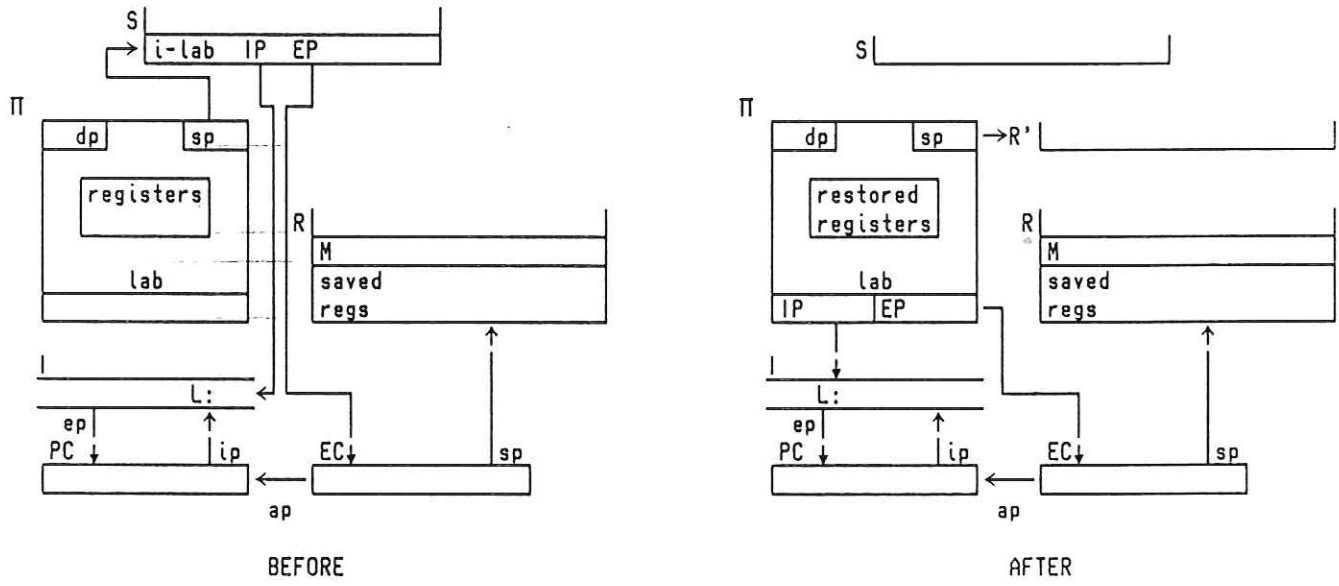
3.8.1. Leaps

A virtual processor which leaps or is coerced to leap (by tele leap) finds its label register loaded with a new instruction label which may cause continuation of execution activity at a site arbitrarily remote from the site of activity obtained just prior to the leap. Leaps must be used carefully and with elaborate preparations since in general both the stack environment and the access environment of a leaping virtual processor are changed by the leap. Since there is in essence one and only one correct stack management strategy (as explained elsewhere), the leap instructions below incorporate a straightforward stack acquisition tactic in support of that strategy; careful coding is required to provide advance preservation of an appropriate stack record to be duplicated by the stack acquisition tactic. By contrast, display vector usages are highly varied and hence the leap instructions below incorporate no display vector actions; display update must be effected by coding at the site of the leap or tele leap instruction.

---

00 leap

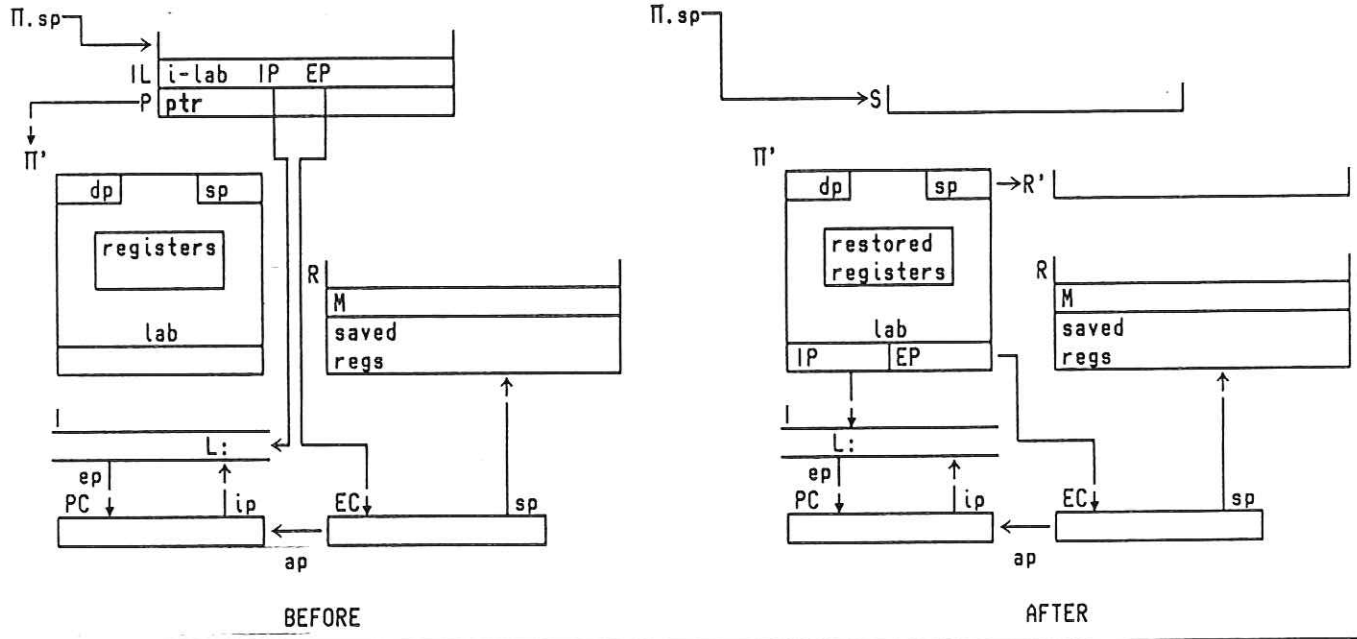
The top record of the stack S must be an instruction label IL whose value consists of an instruction pointer IP and a non-null environment pointer EP which must satisfy the following conditions. IP is a sub-pointerrrof non-ngative index which points to an instuction within some instruction record I whose environment pointer I.ep is non-null and points to some program contour PC; the instruction pointesubrecord PC.ip points to I. EP points to some execution contour EC whose antecedent pointer EC.ap points to PC. The stack pointer subrecord EC.sp of EC is non-null and points to some stack record R which has as its top record a register selector M; below M in R are mono records which have been saved from registers in accordance with the value of M. Execution of this instruction comprises: first, IL is popped from S to a microregister; second, checks are made to ensure that the above conditions are met; third, a duplicate R' of the stack record R is allocated, the reference count of R' is set to 1, and a pointer to R' is placed into the stack pointer register Π.sp of Π; fourth, the steps of res are performed; fifth, a copy of IL is placed into the instruction label register Π.lab of Π.



---

01 tele leap — coerce another virtual processor to leap

The top record of the stack S must be an instruction label IL whose value consists of an instruction pointer IP and a non-null environment pointer EP which must satisfy exactly the same conditions as for the leap (see 3.8.1.00). The next-to-top record of the stack S must be a pointer P to a virtual processor Π' which is necessarily distinct from Π and whose state is either **new** or **asleep**. Execution of this instruction comprises: first, both IL and P are popped from S to a microregisters; second, checks are made to ensure that the above conditions are met; third, a duplicate R' of the stack record R is allocated, the reference count of R' is set to 1, and a pointer to R' is placed into the stack pointer register Π".sp of Π"; fourth, the steps of res are performed **as though by** Π', namely, the saved register contents are popped from R' to the registers of Π' rather then of Π; fifth, a copy of IL is placed into the instruction label register Π'.lab of Π'; the state of Π' is not altered.

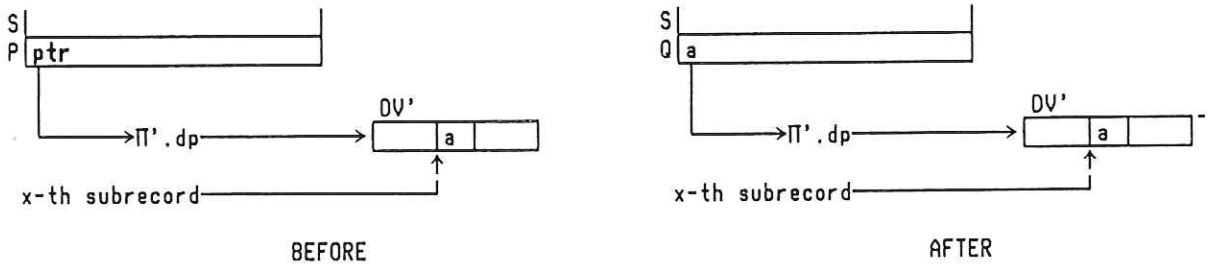


3.8.2. Tele Display Vector Instructions

3.8.2.0. Tele Display Vector Fetch Instructions

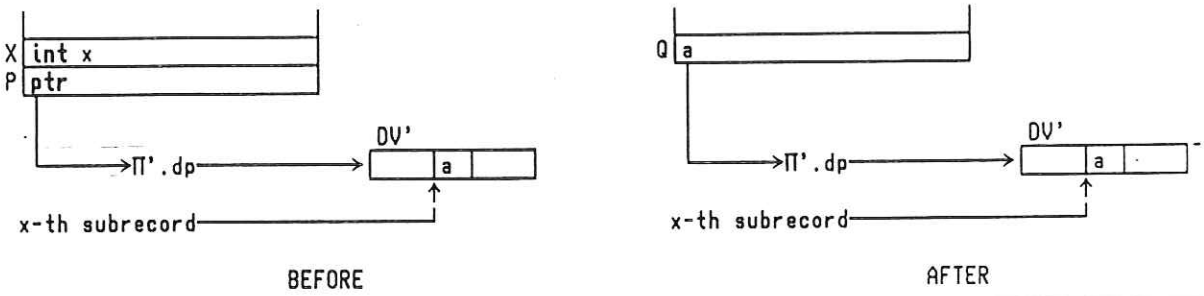
Each of the three one-argument **tele fet dsp** (tele-fetch-from-display-vector) instructions requires two input mono records: first, a pointer P; second, an integer index X = int x. The pointer P must be on the stack S and must point to some virtual processor Π' whose state is either **new** or **asleep**. The integer index X may be an immediate operand in the instruction, may be on the stack S, or may be in some working register Ri, $0 \leq i < 16$. An index which is on the stack S is denoted in the instruction by "*"; stack inputs are popped from the stack prior to formation of the output mono record. If X is an immediate operand then only its value x occurs within the instruction, coded as a short 2's complement field. Each tele fet dsp instruction pushes onto the stack S of Π as its single ouput mono record a copy Q of the x-th subrecord DV'.x of the display vector DV' of Π'.

00 tele fet dsp x — x immediate



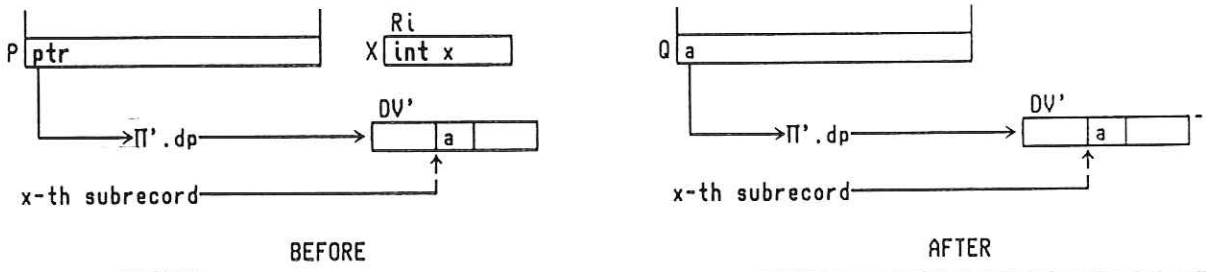
01 tele fet dsp *

x on stack



02 tele fet dsp Ri

x in Ri, 0≤i<16

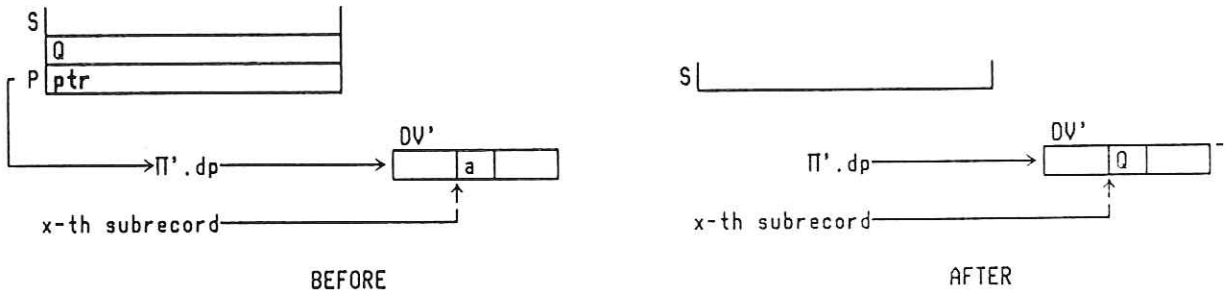


3.8.2.1. Tele Display Vector Store Instructions

Each of the three one-argument **tele sto dsp** (tele-store-into-display-vector) instructions requires three input mono records: first, a pointer P; second, a mono record Q which either is either null or is a (sub)pointer pointing (in)to some execution contour; third, an integer index X = int x. The pointer P must be on the stack S and must point to some virtual processor Π' whose state is either **new** or **asleep** and whose display pointer Π'.dp points to a display vector DV'. The mono record Q must be on the stack S. The integer index X may be an immediate operand in the instruction, may be on the stack S, or may be in some working register Ri, 0≤i<16. An index which is on the stack S is denoted in the instruction by "*"; stack inputs are popped from the stack. If X is an immediate operand then only its value x occurs within the instruction, coded as a short 2's complement field. Each tele sto dsp instruction stores the input record Q into the x-th subrecord DV'.x of th display vector DV' of the virtual processor Π'.

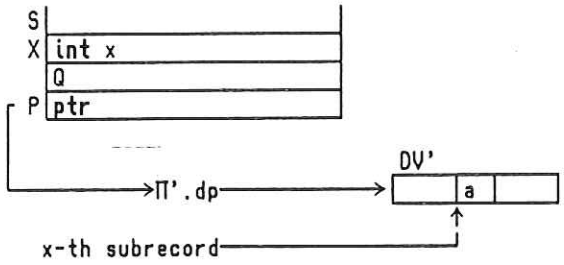
00 tele sto dsp x

x immediate

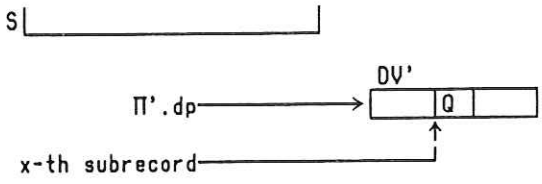


01 tele sto dsp *

x on stack



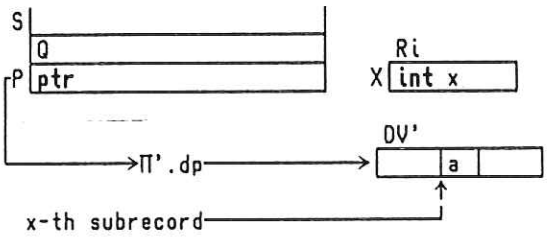
BEFORE



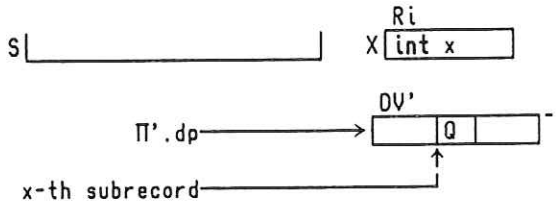
AFTER

02 tele sto dsp Ri

x in Ri, 0≤i<16



BEFORE



AFTER

3.8.3. Module Entry Instructions

Module entry requires specialized instructions both for maintaining the stack and for adjoining new execution contours.

3.8.3.0. Module Entry Stack Instructions

Module entry utilizes specialized instructions both to save registers and exchange label just before entry and to save stack and exchange stack just after entry. To enable the virtual processor Π to coerce another virtual processor Π' to enter a procedure module, tele versions of the instructions to save registers and exchange labels are provided.

00 sav

save selected registers

This instruction is defined in 3.1.1.03.

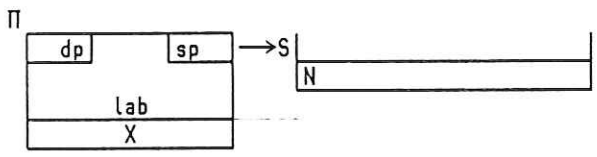
01 tele sav

save selected registers of another virtual processor

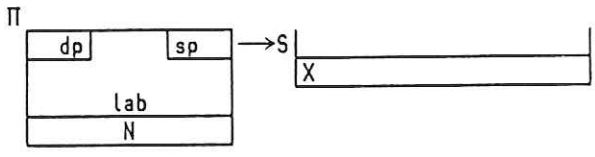
This instruction is defined in 3.1.1.05.

02 xch lab

exchange label



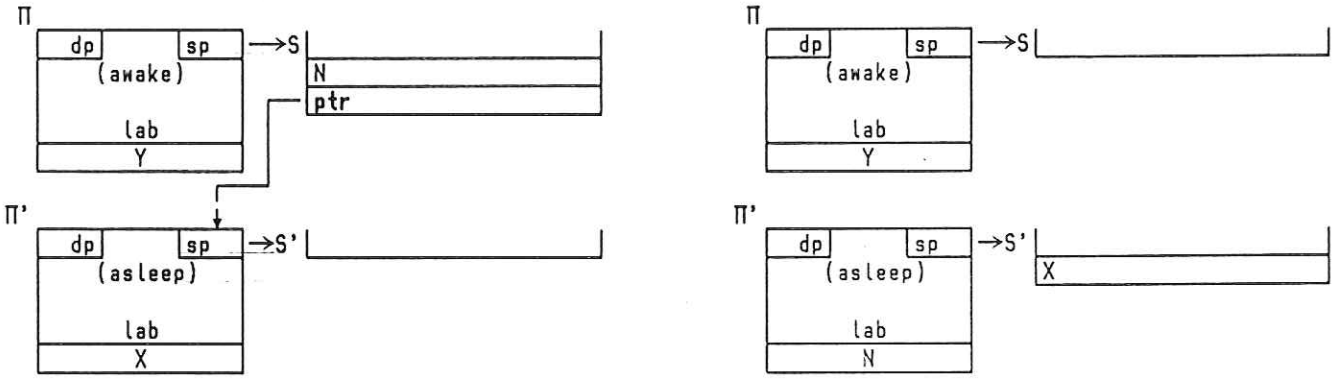
BEFORE



AFTER

The top record N of the stack S must be an instruction label. By the time of the execution phase of this instruction, the instruction pointer portion of the instruction label X of Π has been incremented to point to the instruction in I which immediately follows the exchange label instruction. Execution of this instruction comprises: first, N is popped from S to a microregister; second, X is pushed onto S; third, N is placed into the label register Π.lab.

03 **tele xch lab** exchange label of another virtual processor



BEFORE AFTER

The top record N of the stack S must be an instruction label, while the next-to-top record P of the stack S must be a pointer to another virtual processor Π' whose state is asleep, whose stack pointer points to some stack record S', and whose label register contains some instruction label X. Execution of this instruction comprises: first, N and P are popped from S to a microregisters; second, X is pushed onto the stack record S' of Π'; third, N is placed into the label register Π'.lab of Π'.

04 **sav stk** save duplicate of stack record

This instruction is defined in 3.3.02.

05 **xch stk** exchange stack

This instruction is defined in 3.3.01.

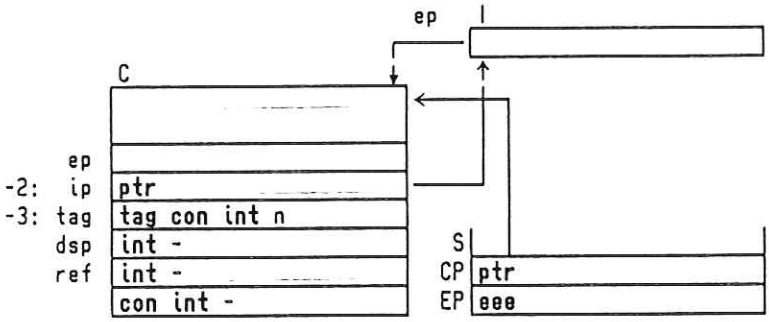
3.8.3.1. The Execution Contour Adjunction Instruction

The execution contour **adjoin** instruction is a powerful instruction which enables the virtual processor Π, given the cp and ep of a contour label designatin both a declaration program module M and an execution environment E for its activation, to perform the following operations as part of the action of establishing the structure required for an entry to be made into M, either by Π or by another virtual processor Π' which is being coerced by Π to enter M: allocate an execution contour of appropriate size, attach that execution contour to the execution skeleton so as to effect the environment E, connect the execution contour to the program skeleton, provide an entry instruction label for entry into M, and provide a pointer to the execution contour to enable the transmission of parameters to the execution contour.

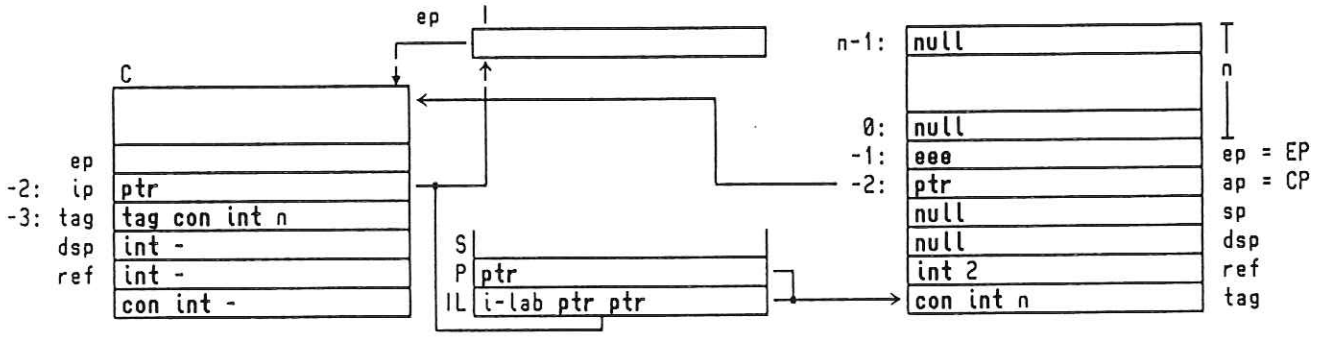
---

00 adjoin allocate and attach execution contour

The top record CP of the stack S must be a pointe to some program contour C, while the next-to-top record EP of the stack S must either be null or be a pointer to some execution contour. Execution of this instruction comprises: first, both CP and EP are popped from the stack S to microregisters; second, an execution contour D is allocated in accordance with the tag subrecord C.tag of C; third, a copy of CP is placed into the antecedent pointer subrecord D.ap of D; fourth, a copy of EP is placed into the environment pointer subrecord D.ep of D; fifth, an entry instruction label whose ep is a pointer to D and whose ip is a copy of the instruction pointer subrecord C.ip of C is pushed onto S; sixth, a pointer to D is pushed onto S.



BEFORE



AFTER

For precision and clarification we give below a pseudo-CMAL coding which is equivalent to the actual microcoded realization of **adjoin**; the registers CPU0 and CPU1 are microregisters distinct from the working egisters R0,...RF.

```
res CPU0; sel CPU0,-3; fet; aloc; res CPU1;          allocate econ
sel CPU1,-1; sto;                                     set e-con.ep
sav CPU0; sel CPU1,-2; sto;                           set e-con.ap
sav CPU1; sel CPU0,-2; fet; mak i-lab;               form entry i-lab
sav CPU1;                                  return pointer to e-con for parameter passage
```

3.8.4. Module Exit Stack Instructions

CMAL provides 26 stack-related instructions for use in conjunction with program module exit. When the virtual processor Π is about to exit a program module M, the conceptual stack CS of Π (see section 2.2) satisfies the following conditions. If M has been entered as a task, then M is a procedure statement module and CS is empty; in all other cases, CS contains at or near its top an exit instruction label EL. If M is a statement module then EL is the top record in CS; if M is an expression module then an exit value EV is the top record in CS and EL is the next-to-top record in CS. If M is a normal program module then CS contains just below EL a register selector RS followed by associated saved register contents; if M is a special system module the register selector and saved contents may be absent. If an exit value EV is present, it is in the stack record S which is attached to Π and constitutes the top stack record of the stack environment of Π. If M is a block module then EL, RS, and saved register contents are in S; if M is a procedure module then EL, RS, and saved register contents are the top records in the next lower stack record S' of the stack environment of Π. Exit from a block module does not require a stack reversion; exit from a procedure module does require a stack reversion and may even require a reversion to a duplicate of the next-to-top stack record of the stack environment. Display vector update may require extraction of the ep of EL for use as a stack parameter to a specialized system procedure for display update.

---

00 rs — revert stack

This instruction is identical to the instruction "rev stk" of 3.1.2.01. Execution of this instruction comprises the following steps. First, a copy SP of the stack pointer subrecord S.sp of the stack record S is placed into a microregister. Second, if SP is null a fault occurs. Suppose that SP is non-null. Third, if SP is not a pointer to a stack record a fault occurs. Suppose that SP is a pointer to a stack record S'. Fourth, if the reference count of S' exceeds 1 a fault occurs. Suppose that the reference count of S' equals 1. Fifth, a copy of SP is placed into the stack pointer register Π.sp of Π.

---

01 cs — conditional duplicate revert stack

Execution of this instruction comprises the following steps. First, a copy SP of the stack pointer subrecord S.sp of the stack record S is placed into a microregister. Second, if SP is null a fault occurs. Suppose that SP is non-null. Third, if SP is not a pointer to a stack record a fault occurs. Suppose that SP is a pointer to a stack record S'. Fourth, the reference count of S'.ref is inspected. If S'.ref equals 1 then a copy of SP is placed into the stack pointer register Π.sp of Π, and the execution is complete. If S'.ref exceeds 1 then an exact duplicate S'' of S' is allocated, the reference count of S'' is set to 1, a pointer to S'' is placed into the stack pointer register Π.sp of Π, and the execution is complete.

---

02 ee — extract environment pointer

Execution of this instruction comprises the following steps. First, if the stack S is empty or if its top record is not an instruction label then a fault occurs. Suppose that the top record of S is an instruction label and that EP is the environment pointer portion of that label. Second, a copy of EP is pushed onto S.

---

03 rl — restore label

Execution of this instruction comprises the following steps. First, if the stack S is empty or if its top record is not an instruction label then a fault occurs. Suppose that the top record of S is an instruction label EL. Second, EL is popped from S and placed into the label register Π.lab of Π. Caution: this instruction is not to be used for an intended leap.

---

04 dl — discard label

Execution of this instruction comprises the following steps. First, if the stack S is empty or if its top record is not an instruction label then a fault occurs. Suppose that the top record of S is an instruction label EL. Second, EL is popped from S.

---

05 rsee — revert stack, extract environment pointer

Execution of this instruction comprises the following steps. First, the steps of "rs" are performed; a fault may occur. Suppose that no fault occurs. Second, the steps of "ee" are performed.

---

06 csee — conditional duplication revert stack, extract environment pointer

Execution of this instruction comprises the following steps. First, the steps of "cs" are performed; a fault may occur. Suppose that no fault occurs. Second, the steps of "ee" are performed.

---

07 rsrl — revert stack, restore label

Execution of this instruction comprises the following steps. First, the steps of "rs" are performed; a fault may occur. Suppose that no fault occurs. Second, the steps of "rl" are performed.

---

08 csrl — conditional duplication revert stack, restore label

Execution of this instruction comprises the following steps. First, the steps of "cs" are performed; a fault may occur. Suppose that no fault occurs. Second, the steps of "rl" are performed.

09 rlrr — restore label, restore registers

Execution of this instruction comprises the following steps. First, the steps of "rl" are performed; a fault may occur. Suppose that no fault occurs. Second, if now the stack S is empty or if its top record is not a register selector then a fault occurs. Suppose that the top record of S is a register selector M. Third, the steps of "res" are performed (see 3.1.1.04).

10 dlrr — discard label, restore registers

Execution of this instruction comprises the following steps. First, the steps of "dl" are performed; a fault may occur. Suppose that no fault occurs. Second, if now the stack S is empty or if its top record is not a register selector then a fault occurs. Suppose that the top record of S is a register selector M. Third, the steps of "res" are performed (see 3.1.1.04).

11 rsrlrr — revert stack, restore label, restore registers

Execution of this instruction comprises the following steps. First, the steps of "rsrl" are performed; a fault may occur. Suppose that no fault occurs. Second, the steps of "res" are performed (see 3.1.1.04).

12 csrlrr — conditional duplicate revert stack, restore label, restore registers

Execution of this instruction comprises the following steps. First, the steps of "csrl" are performed; a fault may occur. Suppose that no fault occurs. Second, the steps of "res" are performed (see 3.1.1.04).

13 rvrs — retain value, revert stack

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "rs" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto the stack record now attached to the virtual processor Π.

14 rvcs — retain value, conditional duplicate revert stack

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "cs" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto the stack record now attached to the virtual processor Π.

15 rvee — retain value, extract environment pointer

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "ee" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto S. Fifth, a "swp" is performed. The actual microcode realization of "rvee" is more effecient than these steps indicate.

16 rvrl — retain value, restore label

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "rl" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto S.

17 rvdl — retain value, discard label

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "dl" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto S.

18 rvrsee — retain value, revert stack, extract environment pointer

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "rsee" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto the stack record now attached to the virtual processor Π. Fifth, a "swp" is performed. The actual microcode realization of "rvrsee" is more efficient than these steps indicate.

19 rvcsee retain value, conditional duplicate revert stack, extract environment pointer

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "csee" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto the stack record now attached to the virtual processor Π. Fifth, a "swp" is performed. The actual microcode realization of "rvcsee" is more efficient than these steps indicate,

20 rvrsrl retain value, revert stack, restore label

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "rsrl" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto the stack record now attached to the virtual processor Π.

21 rvcsrl retain value, conditional duplicate revert stack, restore label

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "csrl" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto the stack record now attached to the virtual processor Π.

22 rvrlrr retain value, restore label, restore registers

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "rlrr" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto S.

23 rvdlrr retain value, restore label, restore registers

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "dlrr" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto S.

24 rvrsrlrr retain value, revert stack, restore label, restore registers

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "rsrlrr" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto the stack record now attached to the virtual processor Π.

25 rvcsrlrr retain value, conditional duplication revert stack, restore label, restore registers

Execution of this instruction comprises the following steps. First, if the stack S is empty then a fault occurs. Suppose that the top record of S is a mono record EV. Second, EV is popped from S to a microregister. Third, the steps of "csrlrr" are performed; a fault may occur. Suppose that no fault occurs. Fourth, a copy of EV is pushed onto the stack record now attached to the virtual processor Π.

A complete listing of the 26 exit stack instructions is given below.

00 rs		13 rvrs	
01 cs		14 rvcs	
02 ee		15 rvee	
03 rl	09 rlrr	16 rvrl	22 rvrlrr
04 dl	10 dlrr	17 rvdl	23 rvdlrr
05 rsee		18 rvrsee	
06 csee		19 rvcsee	
07 rsrl	11 rsrlrr	20 rvrsrl	24 rvrsrlrr
08 csrl	12 csrlrr	21 rvcsrl	25 rvcsrlrr